Mixup: a Development and Runtime Environment for Integration at the Presentation Layer

Jin Yu[1], Boualem Benatallah[1], Fabio Casati[2], Florian Daniel[3], Maristella Matera[3] and Regis Saint-Paul[1]

[1] University of New South Wales, Sydney NSW 2052, Australia
{jyu,boualem,regiss}@cse.unsw.edu.au
[2] University of Trento, Via Sommarive, 14/I-38050, Trento, Italy
casati@dit.unitn.it
[3] Politecnico di Milano, Via Ponzio, 34/5-20133, Milano, Italy
{daniel,matera}@elet.polimi.it

Abstract. In this paper we present a development and runtime environment for creating composite applications by reusing existing *presentation* components. The granularity of components is that of stand-alone modules encapsulating reusable functionalities. The goal is to allow developers to easily create composite applications by combining the components' individual user interfaces.

1 Introduction

User interface (UI) development is one of the most time-consuming tasks in the application development process [3]. As a result, reusing UI components is critical in this process. There is a large body of research in areas such as component-based systems, enterprise application integration and service composition [1], but little work has been done to facilitate integration at the presentation or UI level. While UI development today is facilitated by frameworks (such as Java Swing) providing pre-packaged UI classes such as buttons, menus and the likes, high-level presentation components encapsulating reusable application functionalities have received little attention.

This demo presents a development and runtime environment, called *Mixup*, for integration *at the presentation level*, that is, integration of components by combining their presentation front-ends, rather than their application logic or data. The goal is to be able to quickly build complex user interfaces – and in particular web interfaces – by dragging and dropping existing web UIs (called components) on a canvas and by specifying how components should synchronize based on user and application events. As an example, consider building a US National Park Guide application that includes a park listing, an image displayer showing images given a point of interest and a map displaying the location of a given point of interest. The application can be built out of presentation front-ends such as Google Maps and Flickr.NET[1]. Once integrated, the components should present information in an orchestrated fashion, so that for example when a user selects a national park from the park listing, the image displayer shows an image of the selected park, while the map displays its location (Fig. 1).

In this demo we show how composite applications can be created by combining the front-ends of existing applications (i.e. presentation components), how to define the orchestration logic among components as well as their layout and how all these can be done

[1] The .NET version of Flickr.

quickly via a visual editor. We also show how presentation components can be reused in a variety of composite applications and how an application can swap among components offering similar functionalities.

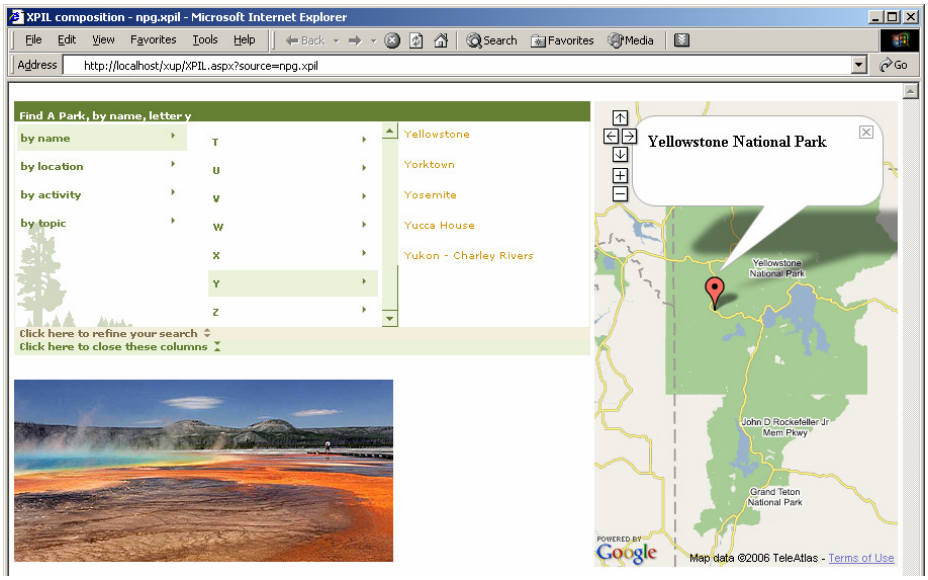

Fig. 1. The National Park Guide

Details on the motivations, rationale, challenges and the proposed approach are provided in [4]. A slideshow of the demo scenario is available on the web[2].

2 The Conceptual Framework

Borrowing lessons from application integration, we argue that integration at the presentation layer needs the definition of four basic elements: component model, composition model, languages and runtime environments.

For the **component model**, we argue that presentation components are centered around the notion of *presentation state*, which is a conceptual, application-specific description of what the component is showing. For example, for a map component, the state may describe the location currently being shown. In addition, a component exposes *events* to notify state changes (e.g. due to user interaction with the component's UI) and *operations* to allow other components to request state changes, so to maintain an overall synchronized UI. Finally, a component has *properties* to represent appearance characteristics (e.g. text color) and customization parameters. Note that the Mixup component model is abstract; that is, it is independent of the technologies used for native component implementations.

The **composition model** allows the specification of event-based integration logics, as we argue that presentation integration is mostly event-based for synchronizing UIs. The integration logics are specified via a set of *event listeners*, each linking an event in one

[2] http://www.cse.unsw.edu.au/~jyu/icwe07/demo.pdf

component to an operation in another component. For example, the park listing component fires a park selection changed event when the user selects a different park; this causes the invocation of an operation on the map component to show a map of the newly selected park and the invocation of an operation on the image displayer to show an image of the new park. For cases where event-based specification is insufficient, additional integration logics may also be specified in the form of simple scripts or external code.

To model components and compositions, Mixup provides a **specification language,** called Extensible Presentation Integration Language (XPIL). It includes a set of XML elements for describing the *component model* (i.e. component descriptor), similarly to WSDL for Web services, and a set of XML elements for specifying the *composition model*. In addition, XPIL also supports the specification of the layout of the resulting composite UI. For example, if the composite application is delivered via the web, the layout information can be specified in HTML and CSS markup.

Finally, a **runtime middleware** executes the resulting composite application by interpreting the specifications in XPIL. In addition, the middleware includes a component adapter framework that allows bindings from the abstract component model to a concrete (native) component implementation. A *component adapter* thus facilitates the communication between the runtime middleware and the native component implemented in a particular component technology (e.g. ActiveX, Java Applet, etc.).

3 Mixup Demo Flow

In this section we demonstrate how the National Park example can be developed and executed using our framework. To create the composite application, the developer follows these steps: i) creating abstract component descriptors (if not yet available) out of UI front-ends of existing applications and save them in a component registry; ii) creating the composite application by specifying its components, their interactions and their layout information; iii) generating the XPIL documents and deploy the composite application to the runtime environment.

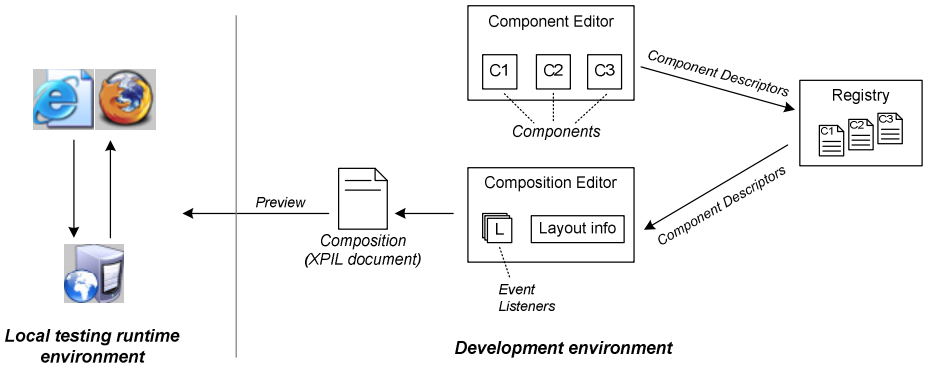

Fig. 2. Development environment

Creating Component Descriptors. The abstract component descriptors for the three components in our National Park example can be created either semi-automatically or manually via the *component editor* (Fig. 2).

First, if a particular component technology supports abstract descriptions (e.g. WSRP) or meta-language facilities such as refection (e.g. Java Applet), the component editor may call a suitable *component inspector* to find out the component's native events, operations and properties and then generate the component model descriptor with the appropriate bindings to the native implementation (this is similar to automatically generating a WSDL document from a Java class). We have developed component inspectors for major component technologies such as .NET Component and Java Applet, since they all support reflection. The component descriptor of Flickr.NET in our example is created by the .NET component inspector.

The component developer may take the automatically generated component descriptor and directly deposit it into the registry. However, typically not all events, operations and properties are needed for the integration. The developer may choose to filter them and to keep only the relevant ones, possibly renamed for better readability.

In cases where meta-language facilities are unavailable, the component editor allows the developer to manually create component descriptors. For the Google Maps component (as well as other AJAX components), the developer can create an abstract operation and specify its binding by pointing to the appropriate JavaScript function.

As depicted in Fig. 3, the editing panel (left hand side of the editor) shows the three components in our National Park example. The yellow flash icon denotes events, and the red rhombus icon denotes operations. Note that the label under the operation or event contains the name of the corresponding native method (e.g. JavaScript function, .NET method) in the component implementation.

Creating Composite Application. To build the National Park example, the developer needs to specify both composition logic and layout information via the *composition editor* (Fig. 2).

First, the developer must select the appropriate components from the registry window (upper right corner of Mixup Editor in Fig. 3) and places them in the editing panel (left hand side of the editor). She then defines event listeners by drawing arrows that link events of one component to operations of other components. For example, the arrow from the "ParkSelectionChanged" event of the park listing component to "showPOI" operation of Google Maps implies that once the park selection is changed by the user, the "showPOI" operation should be called to display the map of the newly selected park. Similarly, the arrow to the "search" operation of Flickr specifies that Flickr should display an image of the park newly selected by the user.

If the event parameters and operation inputs do not match (e.g. number of parameters, data types), the editor will request additional input from the developer. For example, the developer can specify XSLT or XQuery statements that transform the event parameters to operation inputs. In addition, the developer may also specify additional integration and/or transformation logics in the form of scripts or references to external code.

In Fig. 3, the registry window contains several additional components. Specifically, we could use Yahoo Maps instead of Google Maps as the map component in our National Park example. All we need to do is to select Yahoo Maps and drop it into the editing panel and then link the "ParkSelectionChanged" event of the park listing component to the appropriate operation of Yahoo Maps. Similarly, we could also use PBase image service instead of Flickr. Conversely, presentation components can be reused in a variety of composite applications. For example, the Google Maps component defined as per our framework can be used not only in the National Park example, but also in real estate applications and wherever maps are needed as part of the UI functionality.

Finally, the editor includes a layout view for specifying layout information. Non-markup based components (e.g. Java applets, ActiveX controls) are represented by boxes corresponding to their location and size; the developer can move and resize the boxes to define the components' layout information. For markup-based components (e.g. AJAX components), the developer can provide additional CSS statements so that the components can have non-rectangular shapes and can be mixed and overlapped.

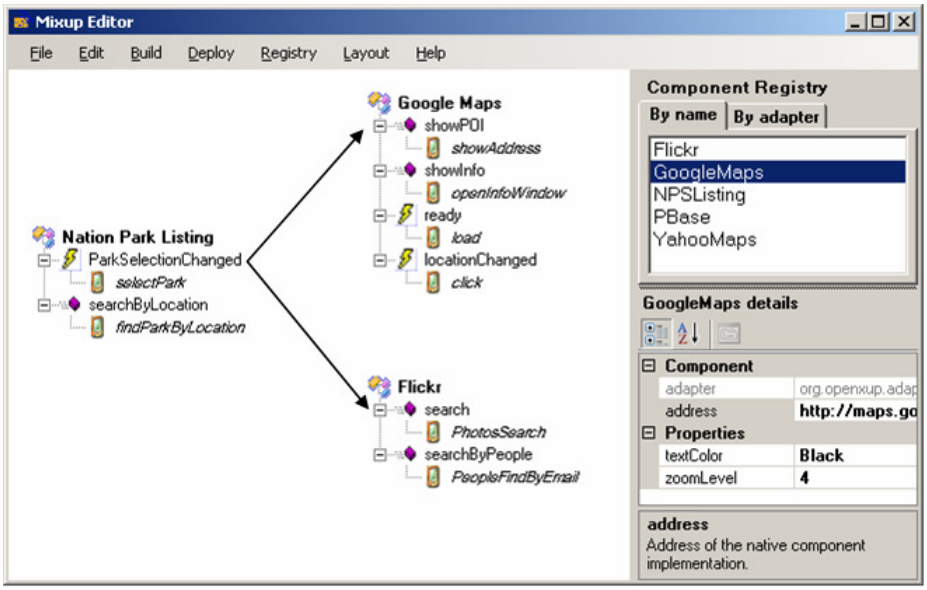

Fig. 3. Mixup Editor

Deployment. Once the component descriptors are created and the composition logic and layout information are fully specified, an XPIL document can be generated and deployed to the runtime middleware. Our development environment includes a testing runtime, which consists of a browser and a web server running in the local development machine (Fig. 2).

Runtime. Finally, the demo shows the integrated UI at work – when the user interacts with one component, the other related components will change in a synchronized fashion according to the specifications in the composition model. The result of executing the composite application, US National Park Guide, is shown in Fig. 1. A detailed description of the runtime framework is provided in [4].

References

1. Alonso, G., et al. *Web Services: concepts, architectures, and applications*. Springer, 2004.
2. Ito, K., Tanaka, Y. A visual environment for dynamic web application composition. *HT'03*.
3. Myers, B. A., Rosson, M. B. Survey on user interface programming. *SIGCHI'92*.
4. Yu, J., et al. A framework for rapid integration of presentation components. *WWW'07*.