Dynamic Composition of Service-Oriented Web User Interfaces

Stefan Pietschmann, Martin Voigt, and Klaus Meißner
Technische Universität Dresden
Faculty of Computer Science, Chair of Multimedia Technology
01062 Dresden, Germany
{Stefan.Pietschmann,Martin.Voigt,Klaus.Meissner}@inf.tu-dresden.de

Abstract

So far, little research has addressed reuse and context-awareness with respect to rich web-based user interfaces. Uniform models for encapsulation and reuse have been developed for the applications’ back ends in the form of Web Services, but this paradigm has not yet been applied to the presentation layer. Thus, UIs are usually hand-crafted and lack flexibility and reuse, which makes their development time- and money-consuming. We address these issues with a system facilitating dynamic, service-oriented composition of user interfaces for modern web applications. UI parts are provided “as-a-service”1 and can thus be selected, customized and exchanged with respect to the current context.

1. Introduction

Lately, the Internet has evolved into a rather stable application platform. The days of static information presentation are long gone – today the WWW provides us with highly interactive applications and services. A shift in application development and deployment from the desktop to the Web can be noted, hence a growing number of applications are provided as “Software-as-a-Service” (SaaS) over the Internet. Grave reasons are location- and time-independent access as well as new business models like pay-per-use.

These new applications are built on top of services which provide data and business logic via generic service interfaces or APIs. We witness that such services are slowly moving towards the presentation layer, which means that they provide UI parts, such as in Google’s Maps1 or Visualization API2. We assume that future web-based applications can be solely based on services that provide either data, business logic or user interfaces. However, current architectural styles do not support such a universal paradigm.

In view of these trends, Web engineering has become a tedious task. Developers are confronted with a myriad of (not necessarily new) programming languages, web frameworks and technologies to choose from. In the end it remains unclear which choice is most future-proof.

At the application back end, solid solutions have been developed for the encapsulation and deployment of functionality – Web Services being the most prominent and widely accepted one. They allow for flexibility of the business logic by exchange and custom configuration of such services. However, there is a lack of comparable efforts for the presentation layer. Current concepts and technologies lack proper reuse mechanisms and interoperability.

A big problem related to this is the maintenance and post-deployment modification of web applications. Despite all the flexibility in the back end, the UI needs to be maintained and adapted to changes manually.

These hand-crafted web UIs usually have a high level of individualization as opposed to classical desktop applications relying on uniform window-based UI libraries. This degree of freedom can be seen as an advantage, however, it often results in inconsistent interaction metaphors, low usability and thus confused users. As there is no standardized concept for reusable UI components for the Web, users need to learn their usage anew in different application contexts.

Finally, to fully exploit the Web’s advantage of time-, location- and device-independent application access, web UIs need to adapt to the current situation, i.e. context (like time zone, location and screen size). However, research in the field of adaptive, context-aware web applications is still largely restricted to basic hypermedia systems and suffers from the “open corpus problem” [1]. Those approaches only work well for closed systems with predefined structures and preindexed or annotated documents, but fail when it comes to Rich Internet Applications (RIA) and unforeseeable, dynamic content. Due to this, the development of modern context-aware web applications is still very cumbersome. Thus, users are yet again confronted with “one-
size-fits-all” RIA solutions, which seems like a step backwards from the achievements of the “Adaptive Hypermedia” community of the last decade.

To address the above-mentioned problems, we strive for a service-oriented UI integration and composition system developed within the project CRUISe3 (Composition of Rich User Interface Services). By using services at the presentation layer as well, we facilitate reuse and flexibility therein and thus simplify the development of context-aware rich web applications.

This paper is structured as follows. In Section 2 we discuss relevant work related to web-based UI component models and UI integration systems. The concept of our integration system, including our UI model, deployment “as-a-service” as well as run time integration, is presented in Section 3. After a brief discussion of the implementation details, Section 4 illustrates the practicability of our approach with the help of an example application. Section 5 concludes this paper and outlines future research directions.

2. Related Work

As described in the last section, the Web lacks uniform models for web-based components as well as related description languages and systems for their composition. We therefore present and discuss related efforts, comprising web component, composition, i.e., integration models and technologies respectively.

First, we need to describe the parts subject to this composition. We define web (UI) components as such software components [3], that provide browser-based means for human-computer interaction. Therefore, the underlying component and composition model is based on web technologies, rather than a programming language.

As discussed in [4], UI integration has not yet undergone much research and has elementary requirements or “key issues”. In short, these comprise a component model, a composition model, inter-component communication styles, discovery and binding as well as visualization mechanisms. In this paper we will discuss all but the fourth requirement, since we leave run time discovery and binding of user interface components as a subject to future work.

One can argue that there already exist numerous component and composition models for the Web. The problem with client-side (JavaScript, Applets, ActiveX Controls, Flash, etc.) as well as server-side (Portlets, ASP.NET Web Parts, etc.) solutions is that they all imply their very own interfaces and communication models. We aim for a uniform approach wrapping everything technology-specific behind a generic interface.

As an example, Portlets are one of the oldest and most mature component models [5]. By composing them within a Web Portal, users are presented a consistent interface of several integrated UI parts including back end services. Thus, Portals constitute both a composition framework and the presentation layer of Service-Oriented Architectures [8, 12]. However, with respect to our requirements, they neither provide a generic, technology-independent component model, nor a standardized component and composition description. Another drawback that is common for the majority of existing solutions is, that UI reconfiguration is only possible by redeployment.

Lately, so-called Widgets or Gadgets have become a prominent and promising approach to web components. However, they constitute rather self-contained applications based on a specific web technology while our definition applies more to interoperable parts of an integrated user interface that communicate via generic interfaces. There is no widely accepted standard specifying a widget component model or description yet, but a movement towards meta-standards can be witnessed, e.g., in W3C Widgets [2] and Google’s Gadgets [6] specifications.

In the context of composite applications Mashups are currently gaining momentum. Their main use is the integration of heterogeneous data, often by means of visual composition by the end user. Therefore, they also handle UI integration to some extent. Composition and integration in these systems is predominantly achieved by custom JavaScript code, which makes development rather time-consuming, or based on proprietary models of mashup platforms or servers.

The “mashup component model” presented in [7] addresses some of these issues in mashup development. It bundles UI-, Action- and Service-Components into a Mashup Component that is subject to composition. Since it is mainly JavaScript-based it remains unclear, if and how other web technologies can be easily integrated. We share the notion of the authors that mashups form an extension of the SOA paradigm. That is why – in contrast to the solution they present – we provide both UI and business logic as services to facilitate maximum flexibility. This allows us to automatically configure and exchange UI parts at run time.

Mürk and Kabanov [9] discuss a system that promotes the integration of different web controller frameworks based on its own object-oriented component model. It is fully Java-based and thus lacks technology-independence and a declarative composition description. As a consequence, the system only promotes design-time integration and does not provide means for the context-aware, dynamic UI reconfiguration that our concept implies.

Finally, the “Presentation Integration Framework” presented in [13] is somewhat similar to our concept when it comes to technology-independence and level of reuse on the user interface. It allows for integration on the component level with the help of a “Composition Middleware”

3http://mmt.inf.tu-dresden.de/cruise/
and platform-specific component adapters. Rather than adapters, we use a generic wrapper that provides platform-independent UI components as a service. Thereby, these components can be distributed and exchanged at run time, while in [13] only design time composition of locally available components is supported.

3. Dynamic, Service-Oriented UI Composition

To overcome the restrictions discussed above, we present a new concept for the development and deployment of web-based UIs. Its central idea is the application of the service-oriented paradigm to the presentation layer to simplify the development of context-aware, service-oriented web applications. By using services to compose a web-based user interface we facilitate reuse, customizability and technology-independence. We do this by (1) the encapsulation of generic, reusable web UI components (UIC), (2) their distributed deployment as so-called User Interface Services (UIS) and (3) their context-aware, dynamic invocation, configuration and integration into a homogeneous, web-based user interface.

After a brief architectural overview, we will present our concept more detailed. First, we specify our generic component model for the encapsulation of UICs. After explaining their deployment as a UIS, we deal with their runtime invocation and integration to a consistent UI.

3.1. Architectural Overview

The overall conceptual architecture of CRUISe shown in Figure 1 conforms to the SOFEA architectural style [11]. This means that presentation logic is not necessarily processed on the server, but rather on the client side.

An integral part of the concept are the User Interface Services – (Web) Services that provide generic, configurable UI components conforming to the model explained in Section 3.2. Initially, an Application Server stores a composition description of an application. When it receives a client request, the corresponding description is processed and requirements as well as configuration parameters of contained UI components are passed to an Integration System. It is responsible for finding UIS in a UIS Registry that match the given application requirements and context. Those UIS are then ranked by their accuracy of fit, the best one being invoked and the resulting, configured UI component returned to the server. Finally, the composed web application UI is sent to the client. There, the event- and data-flow specified in the composition description is controlled by the CRUISe Client component. It manages component interactions as well as communication with remote services that provide business logic and data. Furthermore, context data is sensed and sent to a context management service, which is also used in the discovery and ranking phase.

3.2. User Interface Components (UICs)

Components being part of the composed UI need to adhere to a generic component model. It defines their structure, interface and packaging. Every UIC contains at least a configuration, a content and an interface description.

The configuration description is designed largely along the lines of existing approaches, mainly the W3C Widgets specification [2]. It is an XML-based file that describes all information needed for the integration process, e.g., the name, author information, and, more importantly, references to integrated and external libraries. This information is later used by the Integration System to import the necessary files and initialize the component properly.

The content description forms the “entry point” of the component, i.e. the part to be integrated into the overall UI. It may reference or include other media located within the UIC, such as scripts, styles or multimedia files.

The aforementioned interface description specifies how
to access the UI component (via JavaScript). It defines the UIC entry point needed to initialize and control the UIC, and to facilitate the client-side communication between UICs as specified in the composition description.

Finally, following the W3C Widgets specification, all files are packaged and made available by a User Interface Service, as discussed in the next section.

Due to its openness and simplicity, the majority of components available on the Web (based on JavaScript, Applets/JavaFX, Flash, Silverlight, etc.) can be easily encapsulated with our component model either manually or even (semi)automatically (cf. Section 5).

3.3. User Interface Services (UIS)

User Interface Services are responsible for the distributed and technology-independent provision of User Interface Components. They allow UI designers to deploy and provide individual UI components in a very easy way. Thus, back-end development and UI design can be fully decoupled with the help of services, their combination resulting in a composite application.

Being Web Services technology-wise, UIS are uniquely identifiable by their address and contained version, and can be described with the help of WSDL. To facilitate dynamic service selection, additional UI-specific properties are described in a User Interface Service Description Language (UISDL), which is out of focus of this paper.

Every UIS provides an interface to configure and request a UI component (cf. Section 3.2) and to check its operational availability and status. This interface is later used by the Integration System to retrieve the UIC during the composition phase.

Since the composition is carried out at run time, stability, performance, security and privacy are crucial criteria for UIS to satisfy. To this end, they support the common WS-* standards of the SOAP protocol to meet the above-mentioned requirements.

3.4. CRUISe Integration System

The dynamic UI integration system shown in Figure 2 consists of different modules each responsible for a certain integration aspect. In the following we will discuss each module briefly and illustrate the integration workflow.

Bridge Plug-In When a composite application is requested by a client, the composition description is processed by the application server. Therein, any references to UI services are forwarded by the Bridge to the Integration System, including parameters necessary for the selection and configuration of the required UI component. In contrast to the Integration System, this plug-in is specific to the server-side composition language. In our prototype a JSP-Bridge is used, so as soon as the JSP compiler processes the file, the Bridge redirects all calls for UIS to the Integration System.

Integration Manager Requests for UIS coming from the...
Ranking Strategy

The UIS references and descriptions resulting from the discovery process are ranked based on a particular strategy. It may be exchanged dynamically to allow for domain- or application-specific weightings of ranking criteria. In the end the most suitable UIS is handed to the Invocation Module.

Invocation Module

With the given and contextually resolved parameters from the composition description the eligible UIS is invoked. The UIC returned is decompressed to the file system, assigned a distinct UIS properties available in WSDL and UISDL.

Discovery Module

This component scans available registries – comparable to UDDI – for suitable User Interface Services. Therefore it matches the given application requirements and context data with service and UIS properties available in WSDL and UISDL.

Context Module

Since the selection, configuration and initialization of UIS can be based on the context, this component provides the necessary contextual data, e.g., about the user, client, server or application. The quality and amount of this information heavily depends on the underlying context monitoring and modeling system. In [10] we have already presented a suitable, service-based solution. The Context Module primarily replaces contextual placeholders with actual data and evaluates context conditions.

Integration Plug-In

Finally, the UIC component in question is integrated into the markup language used in the composition document. Furthermore, the wiring between UI components as specified therein it realized. In most cases HTML is the best choice as “glue” between components of different nature, since it is the common ground for all browser-based technologies. However, by using another Integration Plug-In, we can support any other declarative language, such as Flex or XAML, and thus compose a Flash or Silverlight application out of UIS with the help of our system.

Once the Integration Task is finished, the Manager returns markup code integrating and initializing the UIC back to the application server, where it is woven into the composite application description. Once the description is fully processed, the final application is sent to the client.

3.5. Client-Side Components

When the composite application is executed in the browser, the CRUISe Client Module manages the underlying communication between UI components based on a publish/subscribe mechanism. Therefore, the JavaScript interface mentioned in Section 3.2 is used. The event and data flow follows the rules specified by the author in the composition description. Additionally, the Client Module is responsible for sensing client and device capabilities as well as monitoring user interactions to provide the context monitoring service with this information.

4. Implementation

To verify the concepts presented in the previous section the CRUISe Integration System was implemented and tested with the help of an exemplary composite application.

To this end, a number of User Interface Services were developed, encapsulating different well-known components, e.g. a Google Map. Their Web Service functionality is largely based on Apache Axis2 and thus benefits from steady improvements and comprehensive standards support. Besides SOAP being established in SOA environments, MTOM is used for an efficient, optimized data transfer of UI components to the service consumer. Each UIC is given a specific namespace to allow for unique identification of unpacked components. To assure uniqueness of class and object identifiers therein (with respect to the composite application), we use so-called Universally Unique Identifiers (RFC4122) that are replaced with distinct identifiers during the integration process.

The Integration System was realized in Java according to the architecture illustrated in Figure 2 with some additional functionality like local resource management and caching. We focussed on component integration rather than discovery and contextualization, the latter two being subject to future work. For the prototype we implemented an Integration Plug-In for HTML based on the integration mechanism of the jMaki framework. It realizes unique identifiers as mentioned above, wraps components in HTML containers and embeds the CRUISe Client Module into the application.

The Client Module is based on jMaki’s client-side controller and uses the jQuery library. Besides the publish/subscribe mechanism controlling the event flow, it provides means for users to adapt the UI to some extent, comparable to Portlets (e.g., by maximizing or minimizing parts). We plan to monitor those interactions in order to improve the users’ context models.

References:

1 Apache Axis2: http://ws.apache.org/axis2/
2 W3C MTOM: http://www.w3.org/TR/soap12-mtom/
4 jMaki: https://ajax.dev.java.net/
For demonstration and testing purposes, a prototypical, composite “contacts management” web application and the corresponding JSP-Bridge were built. As can be seen from Figure 3 it allows for a simple management of contacts and provides additional information on their current location. Users may edit their information with the help of a form or by changing their location directly on a map. The UICs developed range from simple HTML and JavaScript (Google Maps, Dojo) to Flash (Flex) and Google GWT components. Our prototype exemplifies the data binding with the back end (contacts are provided by the application), the communication between UI parts as well as the technology independence regarding the UI components used.

Figure 3. Prototype integrating several UIS

5. Conclusion and Future Work

The development of flexible, context-aware user interfaces for future web applications is time- and money-consuming. In this paper we have presented an approach for the service-oriented and context-aware composition of rich web UIs. Particularly, we have specified a generic model for configurable web components, a corresponding service-based deployment method, and a run time system for the dynamic, context-based discovery, selection, configuration and integration of these “User Interface Services”. To validate our approach, we have built a prototype illustrating the dynamic composition of the UI from components of different technologies, and their two-way communication via a generic interface.

To our knowledge, CRUISe marks the first service-oriented approach to UI composition. It greatly simplifies development, reuse and maintenance of web-based user interfaces by deploying them “as-a-service”. Beyond this, it allows for greater flexibility and context-awareness on the presentation layer of modern rich Internet applications.

Currently, we are working on a generic wrapper component that facilitates the generation of UIS from arbitrary web components (JavaScript, Flash, Silverlight, JavaFX) to improve the toolchain support. To allow for the context-aware exchange and configuration of UIS, we plan to provide both a classification and description language for them, as well as adequate ranking mechanisms to support the decision at run time.

6. Acknowledgements

The CRUISe project is funded with means of the German Federal Ministry of Education and Research under promotional reference number 01IS08034-C.

References