Platform Adaptation of Mashup UI Components

Andreas Rümpel, Ken Baumgärtel, and Klaus Meißner
Chair of Multimedia Technology
Technische Universität Dresden
01062 Dresden, Germany
{*andreas.ruempel,ken.baumgaertel,klaus.meissner*}*@tu-dresden.de*

***Abstract*—Modern *user interface mashups* combine web-based backend services and a composite user interface. While the former can be used in different runtime platforms without further ado, user interface components are rendered platform-dependently. Creating and maintaining multiple implementation variants for different technologies and communication interfaces implies high development costs. This paper introduces a generic platform adaptation concept of such visual mashup components. The practicability of the proposed concept is demonstrated by implementing the adapter for a subsistent integration infrastructure and component model.**

***Keywords*-user interface integration; platform adaptation; mashup components; composite web applications**

I. Introduction and Motivation

Old-fashioned ways to create web applications presupposed a specific execution platform or framework. This includes server-centric ones, like portal servers, those running completely on the web client, e. g., following the *Thin Server Architecture* [1], and hybrid ones like *Eclipse Rich Ajax Platform (RAP)* [2]. When building composite web applications for those platforms, the availability of compatible UI components is essential. Such UI components cannot be reused in other runtime environments easily, because different web platforms and runtime frameworks have specific technology-induced requirements and behavior concerning their structure, language and deployment mode and thus impede a cost-efficient and fast development process. This platform heterogeneity implies a tedious manual adjustment and migration process of existing components to be executed in alternative environments. We understand these runtime platforms as one special kind of context called *integration context*. Supporting various integration contexts can be regarded as a new quality of adaptivity of such applications, driven by their components.

In most composite systems, *component interface descriptions* represent the formal realization of a *component model*. They are used to facilitate the application's internal communication. The existence of such component interface descriptions is an essential precondition of our adaptation concept. In contrast to conventional web-based services, user interface building parts apparently cannot be platform-independent per se, because they are always rendered based on concrete code, i. e., they mostly contain or generate HTML code in conjunction with JavaScript. We define *platforms* including web runtime architectures and UI toolkits as well, which are already present in a great diversity. This indicates that a platform adaptation mechanism is inevitable against the background of using UI components in a multi-platform context. Fortunately, the major part of one UI component can be generated automatically, e. g., through derivation from an interface description during platform adaptation.

This paper provides a novel concept introducing a platform adapter for UI components of composite web applications. It takes existing user interface components conforming to a specific component model and developed for a specific runtime environment as adaptation input. Based on templates, it performs platform adaptation by wrapping and generating the parts relevant to the target runtime platform.

The remaining paper is structured as follows. Section II analyzes and defines prerequisites regarding the component model in distributed web UI scenarios. Section III presents our concept of platform adaptation for UI components. Afterwards, in Section IV, the proposed concept is applied to the CRUISe UI composition infrastructure [3], which was developed in parallel. Implementation details are given in Section V. In Section VI, related work is outlined. Finally, Section VII concludes this paper.

II. Component Model and Prerequisites

We consider UI components in the scope of web-based composite applications, causing the need for a client-side part to be rendered by a web browser engine. This client-side part, generically created or not, consists either of HTML and JavaScript, or employs a browser plug-in technology like Adobe Flash or Microsoft Silverlight. To specify an adaptation concept, a discussion of valid combinations of source UI component implementation architectures and target platforms is necessary. The following comparison elucidates different commonly used UI component implementation techniques and derived consequences for the adaptation process with special focus on client-server distribution.

Client-side JavaScript: Components using only plain HTML and JavaScript frameworks are well-suited to be adapted to other JavaScript frameworks, hybrid or server-side technologies because of their easy to handle declarative

```
<interface xmlns:dt="http://uis.dyndns.org/datatypes">
   <event name="singleSelection">
      <parameter type="dt:Address" name="Address"/>
   </event>
   <event name="multiSelection">
      <parameter type="dt:AddressList" name="AddressList"/>
   </event>
   <event name="exportSelected">
      <parameter type="dt:AddressList" name="AddressList"/>
   </event>
   <operation name="addRecord">
      <parameter type="dt:Address" name="Address"/>
   </operation>
   <operation name="updateRecord">
      <parameter type="dt:Address" name="Address"/>
   </operation>
</interface>
```

Listing 1. Interface description of an address list UI component

and scripted nature. In the latter two cases, additional component parts on the server side have to be generated.

Browser plug-in: Adobe Flash, Microsoft Silverlight and other browser plug-in technologies are also client-side approaches. They are scriptable using a JavaScript bridge and then can be adapted to the same as above.

Hybrid techniques: They are suitable to be adapted under certain conditions, if the UI-relevant part can be extracted like in owner-drawn RAP widgets [4]. In this case, the output format range is like the above.

Server-side (binary and scripted): Server-centric component implementations require a code analysis of source components, if technologies like portlets are used and an extraction technique for server-side scripting approaches, like Java Server Pages. Possibilities supporting server-side input components are very limited, because binary components or component parts would have to be provided as source code.

Obviously, the ability of handling implementations of available source components is a crucial factor when performing component adaptation. An important cornerstone of our adaptation concept is the use of a *component model*, formally describing the communication interfaces of components. To exemplarily illustrate the contents of such a model, a snippet of a mashup component description used in CRUISe (cf. Section IV) is shown in Listing 1. It describes the interface of an *AddressList* UI component. The interface model used here comprises *events* and *operations* to establish an event-based communication within the application. Three events (*singleSelection*, *multiSelection* and *exportSelected*) and two operations (*addRecord* and *updateRecord*) with corresponding data types are provided by this component. Using those interface parts, the addresses can be read out or manipulated.

In Figure 1, the explained component is shown in a simple real estate management application context. New addresses can be added to **AddressList** (middle, same interface as outlined before) by another component, e. g., a **Building-Database** (left), providing new address entries using the operation *addRecord*. Once an address entry is selected in the list (*singleSelection*), the building corresponding to the selected address is visualized by an **ImageViewer** component (right). The whole communication (arrows) is realized by an event-based interface in a publish-subscribe fashion.

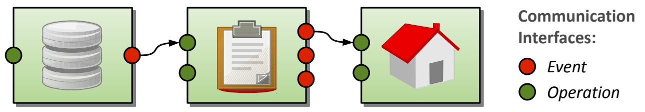


Figure 1. User interface components in a mashup application

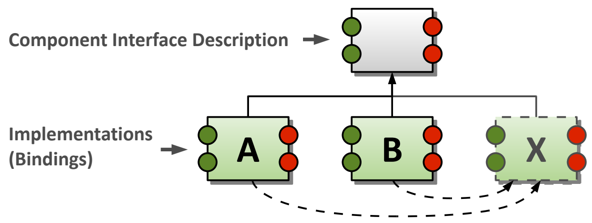


Figure 2. Adaptation as derivation of new UI component binding

We conceive component implementations as *bindings* for an abstract component interface description. Accordingly, different bindings of the same interface are exchangeable against each other within an application. Thus, the proposed platform adapter can be regarded as a generator for new bindings, matching the previously unsupported integration context to adapt to. As illustrated in Figure 2, an existing binding (e. g., *A*) for another integration context is taken as a source of adaptation to derive the new one (*X*) utilizing platform-specific creation rules (arrows). These creation rules have to be specified once per platform to be supported.

III. UI Component Platform Adapter

Based on the previously stated prerequisites and conditions, we now present our concept of platform adaptation of UI components proposing a multi-staged platform adapter to be used within web integration architectures. The adaptation workflow is executed for one UI component at a time. Starting point and input for the adapter are the component's abstract *interface description* and one of its *bindings*. As shown in Figure 3, an abstraction stage precedes the template-based generation of the target implementation to transform non-matching input bindings into a platform-compatible one. The service, the platform adapter offers to the UI integration system, can be summarized as the provision of a previously unavailable binding for a specified UI component. Beside a list of source bindings and the component's interface description, the adapter requires an identifier of the desired target platform as input. While the support of different platforms mainly affects the second adaptation stage, the used abstract component interface model has to be specified for the whole adapter. Hence, different component models employing alternative communication paradigms are fine, as long as they conform to the outlined prerequisites (cf. Section II). The two stages are now described in detail.

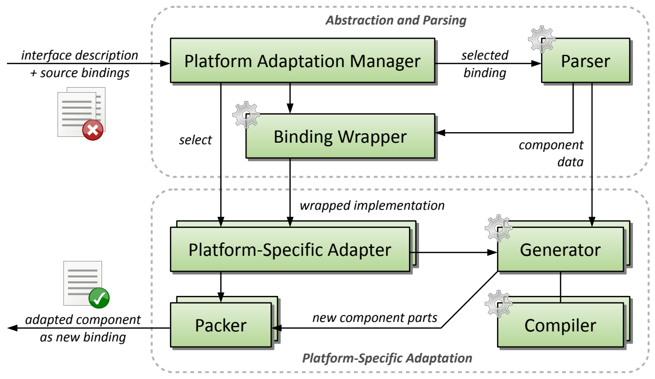


Figure 3. Platform adapter architecture

```
<binding platform="javascript-client-runtime-platform">
   <interface>http://uis.dyndns.org/uic?class=addresslist</interface>
   <dependencies>
      <dependency uri="http://uis.dyndns.org/addresslist.js" language="js"/>
   </dependencies>
   <constructor>@:instance@ = new ui.component.AddressList()</constructor>
   <destructor>@:instance@.remove()</destructor>

   <eventsink event="event.singleSelection">
      <register>@:instance@.on('singleSelection',
               @event:event.singleSelection@)</register>
   </eventsink>
   <eventsink event="event.multiSelection">
      <register>@:instance@.on('multiSelection',
               @event:event.multiSelection@)</register>
   </eventsink>
   <eventsink event="event.exportSelected">
      <register>@:instance@.on('exportSelected',
               @event:event.exportSelected@)</register>
   </eventsink>
   <invocation operation="op.addRecord">
               @:instance@.addRecord(@parameter:Address@)</invocation>
   <invocation operation="op.updateRecord">
               @:instance@.updateRecord(@parameter:Address@)</invocation>
</binding>
```

Listing 2. Binding for a JavaScript *AddressList* component

A. *Abstraction and Parsing*

In this stage, the analysis of the components' interface description and the derivation of an internal data model of the input UI component binding are performed. The *Platform Adaptation Manager (PAM)* controls the whole adaptation process. In doing so, it selects one of a list of available source bindings to proceed further steps. As shown in Listing 2, bindings are represented by an XML description for a specific `platform`, referencing needed `dependencies` and containing the mapping to language-specific constructs, as the JavaScript snippets illustrate. The depicted binding conforms to the *AddressList* component (cf. Listing 1), telling how the implementation should be addressed through `invocation` of *operations* and `eventsinks` for publishing *events*.

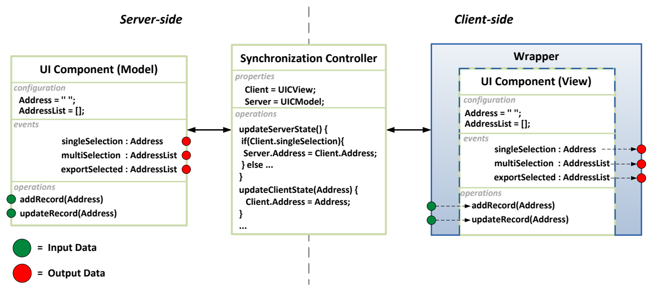
The binding is analyzed by a *parser* to create an internal interface model, which is forwarded to the *binding wrapper*. Its task includes the resolution and download of the component's dependencies. This step is necessary, because components differ in their style of referencing and including resources, such as images, styling information or script files. Based on the collected component data, a common representation for further processing in the second adaptation stage is gained.

B. *Platform-Specific Adaptation*

The wrapped component implementation represents the input for a *Platform-Specific Adapter (PSA)*, which is selected by the *PAM* depending on the target platform identifier specified. A concrete PSA controls the *template-based* creation of all parts of the new UI component implementation. Different *generators* and *compilers* are invoked by the PSA to perform this task. The complexity of implementation artifacts to be generated depends on the target platform (cf. Section II). Having all parts generated, a *packer* brings them into a deliverable format, creates an XML file describing the binding declaratively (cf. Listing 2) and links resources within the binding document.

For supporting a large range of platforms, a flexible internal component handling is required. The adaptation to a client-side platform considers inter-component communication and integration on the client, while a server-side platform performs this on the server. Hence, the adapted UI component is divided into different parts along the lines of a design pattern called *Half-Object plus Protocol (HOPP)* [5]. This division enables the partitioning of provided functionality of the UI component depending on the target platform and introduces the principle of *Model View Controller (MVC)* meeting the demands of adaptable UI components. With this conceptual division, each part of the component can be adapted effectively, so that it will fit best into the target platform. If, e. g., a component for a client-server distributed integration context with server-side communication is targeted, HOPP helps to build a compatible server-side interface and to connect it with the corresponding client-side part of the component. Figure 4 shows the distributed nature of the *AddressList* component outlined in Listings 1 and 2 for a client-server distributed integration context. The server-side part represents the model, the client-side part the view and another part, called *Synchronization Controller*, adopts the implementation of the used protocol and represents the controller, or bridge, between both other parts. This distribution causes the server-side part holding a reference to the wrapped, previously client-side UI component and is therefore a *proxy*. The input (green) and output (red) access points enable component communication. This communication can also be parallel, on client and server, yielding UI components enabled to communicate offline, synchronizing their corresponding parts and persisting their states when a network connection is established.

The splitting, i. e., distribution of the component, is not mandatory. It depends on the given component technology and the architecture of the target platform. Thus, it is possible to have only a client-side wrapper (e. g., for a thin server runtime) without the other parts. Assuming a distributed



```
Client = UICView;
Server = UICModel;
```

```
updateServerState() {
 if(Client.singleSelection){
  Server.Address = Client.Address;
 } else ...
}
updateClientState(Address) {
  Client.Address = Address;
}
...
```

Figure 4. Distributed adapted UI component for a thin client platform

integration context, like Eclipse RAP [2], a distributed component is generated. It consists of a package containing the server-side model, the client-side JavaScript component and the so called *Life Cycle Adapter* to synchronize both. In other situations, e. g., a JavaScript-based integration context, it is sufficient to create a JavaScript wrapper. Thus, a suitable workflow is achieved, that covers many platform-specific possibilities.

C. Adapter Usage and Integration

At least one proper PSA per target platform has to be provided to make use of this adapter. Therefore, it is the PAM's responsibility to choose the right PSA at runtime. To develop Platform-Specific Adapters, adaptation experts require a high knowledge of the integration contexts to cover. If there is no adequate PSA available, the adaptation is aborted and the PAM cannot deliver a new binding. Runtime platform providers are in charge of creating and maintaining their PSAs. The adaptation process can be called at application runtime, at UI integration time (cf. Section IV), or at component deployment time in a component registry, which denotes a higher performance. Moreover, the adapter itself could be outsourced and act as an external service, the integration system could use. A similar way of modularization could be applied to the PSA to yield *Platform Adapation as a Service*. This would achieve the total separation from an explicit integration infrastructure.

IV. Service-Based UI Integration System

The *CRUISe system* [3] facilitates the integration of service-based user interface building parts, called *User Interface Services (UIS)*, to create *user interface mashups*. This section describes how CRUISe is utilized to provide a UI integration infrastructure hosting the platform adapter.

A. Architectural Overview

As shown in Figure 5, using a platform-independent *composition model* and a model-to-code transformation, one application can be executed on different kinds of distributed runtime platforms. The composition model is transformed into a platform-specific composite CRUISe application (green arrow). Within this model, UI component

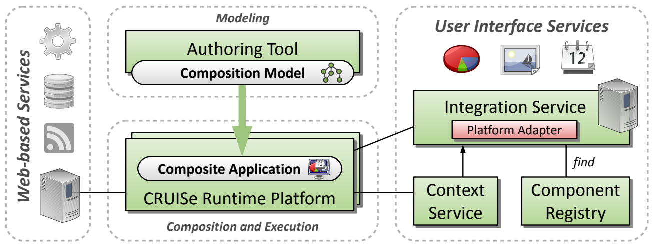


Figure 5. CRUISe infrastructure

interfaces and their communication relations are specified. The generated platform-specific application contains binding points for UI components, enabling binding and exchange at runtime. Thus, it is called *application skeleton*. The interface descriptions of the components to be integrated are organized in classes to modularize them and facilitate the implementation of multiple bindings. Referencing such interface classes (cf. interface description in Listing 1), all information needed to integrate UI components in a service-oriented fashion is provided. Taking those descriptions and a platform identifier, an integration request is created and sent from the runtime platform to the *Integration Service*, which performs further steps of retrieval, selection and adaptation, utilizing the *Component Registry* and external *Context Services*. The ready-to-integrate UI component is then returned to the runtime environment. A crucial condition for this workflow is the availability of platform-compatible component implementations, i. e., bindings, to be delivered. Hence, only existing bindings can be provided at this point without a platform adapter. Since the binding selection and ranking process is executed with the *Integration Service* in charge, our adapter is realized there as a module.

B. Integration Workflow

An integration request is sent from the runtime environment to the *Integration Service* carrying component interface descriptions and a platform identifier. Interface descriptions are extracted from the application skeleton, the platform is determined by the runtime platform itself based on a defined set of identifiers. By executing an *Integration Task*, this request is forwarded to the *Component Registry*, i. e., it is asked for matching UI component implementations of the UI component, that is subject to integration. The further workflow can be divided into two cases.

Platform binding available: The registry discovers bindings matching the component interface and the platform. Then a list of compatible bindings is returned to the *Integration Service* and no platform adaptation is needed. Next, a ranking module performs a context-based ranking process, yielding the most appropriate UI component in respect of the given requirements.

Platform binding unavailable: The registry can find bindings matching the interface description, but not the platform parameter. As a consequence, the registry sends back an empty list. Next, the integration service sends a new request, but without restricting the platform. A list of bindings is returned, which are not matching the initially desired platform. Afterwards, the *Integration Service*, knowing that these bindings are not platform-compatible, can continue to forward this list to the platform adapter. There, the *Platform Adaptation Manager* receives the list and performs further operations described in Section III. The employment of a ranking module is obviously not needed in that case.

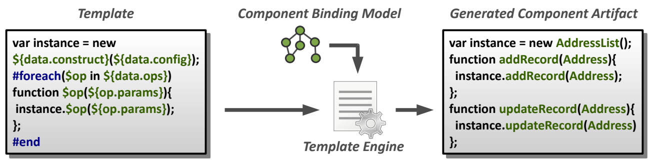
Thus, platform adaptation is added as a further step to the CRUISe integration task. Its invocation is depending on the output bindings of the *Component Registry*. In both cases, the finally selected binding is returned to the runtime environment, which performs its integration into the composite application. The new binding may comprise different new files, that were created during adaptation. To allow the runtime environment to access them, they have to be hosted under a specific address, which is referenced within the binding XML document. This kind of hosting is provided by the *Integration Service*.

While processing this integration workflow, some exceptional situations could occur. It is possible that the registry cannot find any bindings, even when neglecting the platform parameter, but then a UI component without implementations would exist. This situation should be avoided by the component developers. Incorporating the proposed concept, the model-based development and UI integration approach in CRUISe facilitates the execution of one mashup application in different runtime platforms.

V. Implementation

Since a concrete component model is required to implement the platform adapter, the CRUISe-specific model was chosen to fit into the integration infrastructure described in Section IV. For the template-based component part generation we used *Apache Velocity* [6] as a template engine. Templates are provided with the selected PSA for each supported target platform. Each template contains *place holders*, using *Velocity Template Language*. They are filled by a *template engine* with information from the extracted component data (Figure 6). A place holder defines a variable part in the template. It is expressed by the $-symbol for accessing data objects and optionally embedded in statements given by the #-symbol, e. g., to handle loops. With both constructs, each template can be described effectively. In Figure 6, a new instance of the *AddressList* component and the definition of the component's operations are represented by place holders (cf. Listing 2).

The step of generating the component's parts includes the download of and reference to source component artifacts, gained from the binding description, and the optional compilation of created source code after template processing. If, e. g., a component part based on Java should be created, the



```
var instance = new
${data.construct}(${data.config});
#foreach($op in ${data.ops})
function $op(${op.params}){
 instance.$op(${op.params});
};
#end
```

```
var instance = new AddressList();
function addRecord(Address){
 instance.addRecord(Address);
};
function updateRecord(Address){
 instance.updateRecord(Address)
};
```

Figure 6. Template-based component artifact generation

PSA would call a Java compiler to generate bytecode out of the Java source code. For such typed programming languages, the right communication data types in the templates have to be defined. The used component model defines data types with XML Schema. Finally, an additional generation step of the target component structure and packaging to the target format is executed. In the case of RAP, it is packed as a *Java Archive* to provide an *OSGi bundle*. An RAP-based runtime can load those bundled components and integrate them into an RAP application. Hence, the package contains artifacts of the original UI component, its resource dependencies, like optional CSS, Flash or JavaScript files, wrapping code and the new XML binding description, generated by a special template. Finally, the new binding is returned and can be committed to the hosting integration infrastructure.

VI. Related Work

While the attention for mashup development has been overwhelming, only few preliminary work has been done in the field of mashup component adaptation. Often, a sufficiently large pool of perfectly matching UI components is assumed, but compatibility issues initiated by different component models and runtime platforms are neglected.

In [7], a wrapping mechanism of generic web applications is described by using a predefined component model to provide them for further composition. The transformation of web applications is done using *event annotations* to yield mashup components conforming to the component model. A generic wrapper structure enables the support of componentization at runtime. An API is created, which generates events, enacts operations, allows the instantiation and renders the component's UI. In contrast to our approach, they propose the engineering of new components by using existing web pages. No source component model is presumed. The goal is to create new mashup components for further composition. This process is performed by a component developer with a special tool, while our adapter creates new component implementations automatically, showing the huge relevance of a predefined component model.

A similar approach of adapting existing UI building parts is also used in the *CAMELEON* project [8]. It identifies three steps to re-engineer a UI to other contexts. Multiple levels of abstraction provide the step-wise process of generation [9]. First of all, UI interaction objects of a web page are detected, while a user provides feedback on the objects he is interested in. Next, a presentation model is created out of them. After that, the model is transformed into another presentation model of the new context. It can be manually modified before creating the final user interface. In contrast to CAMELEON, we strive for an automatic process without any user feedback at runtime. Furthermore, it is hardly applicable to analyze a UI without the knowledge of its component model. Thus, the kind of component introspection and re-engineering like in CAMELEON is not suitable in our case. Further, to support a service-oriented UI provision, parsing is only applicable at the level of component interface description.

Mixup elaborates the usage of component adapters for the integration of presentation components [10]. They aim for the support of heterogeneous components with different underlying technologies. A lightweight middleware is used to instantiate these adapters to allow the communication of UI components at runtime. The adapter locates the right implementation and instantiates the component. It identifies and allows access to events, operations and properties and performs data type mapping. To apply this adapter concept, a meta-language facility like reflection is needed. If this is not satisfied, a generic adapter cannot be built. In that case, the implementation of individual wrappers for each component is needed. We also use a component wrapper, which is part of the adapted UI component itself. For creating this wrapper we use an abstract interface description of the component as an input. This is very important for encouraging the independence of possible runtime systems as the adapter does not need to be part of them.

VII. Conclusion and Future Work

In this paper, we presented a platform adaptation concept for UI components, which are used within web mashup applications. Therefore, we took the existence of a *component model* as a starting point and treated component implementations as *bindings* to their interface descriptions realizing this model. We regarded the platform adaptation process as a template-based generation of further bindings to a given component interface supporting different runtime platforms. By transferring the platform adapter to an existing user interface integration infrastructure, we showed the operational capability of our concept, implementing *Platform-Specific Adapters*, e. g., for a runtime environment based on Eclipse Rich Ajax Platform. All in all, we are able to employ web UI components in a multi-platform scope.

The current approach does not automatically involve similarities between different integration contexts. To this end, a classification of integration contexts could help to derive PSA implementations covering new platforms. Thus, a semi-automatic adaptation workflow for unknown platforms similar to known ones would be possible, being a very useful tooling support for platform refactoring and version management. We further plan to investigate how generated bindings, i. e., adapted components, can be saved and then made available to a component repository to be used in further integration tasks. An additional research subject will cover the extension of the UI component adaptation concept to a general mashup component adaptation concept. Prerequisites are a uniform mashup component model and analyses of the adaptation and exchangeability needs of components such as data type converters or service proxies.

Acknowledgements

The authors wish to thank CAS Software AG for providing support by developing a prototypical RAP-based runtime platform to be used as an integration context.

References

[1] S. Pietschmann, J. Waltsgott, and K. Meißner, “A thin-server runtime platform for composite web applications,” in *Proceedings of the 5th International Conference on Internet and Web Applications and Services (ICIW 2010)*. IEEE CPS, May 2010.

[2] B. Muskalla and R. Sternberg, “RCP goes web 2.0,” *Eclipse Magazin*, vol. 12, Oct. 2007.

[3] S. Pietschmann, M. Voigt, A. Rümpel, and K. Meißner, “CRUISe: Composition of rich user interface services,” in *Web Engineering*, ser. Lecture Notes in Computer Science, vol. 5648/2009. Springer, Jun. 2009, pp. 473–476.

[4] F. Lange, *Eclipse Rich Ajax Platform: Bringing Rich Clients to the Web*. Apress, Dec. 2008.

[5] G. Meszaros, “Pattern: Half-object plus protocol (HOPP),” in *Pattern languages of program design*, J. O. Coplien and D. Schmidt, Eds. Addison-Wesley Longman, May 1995, pp. 129–132.

[6] Apache Software Foundation, “The apache velocity project,” accessed: 2010-08-26. [Online]. Available: http://velocity.apache.org

[7] F. Daniel and M. Matera, “Turning web applications into mashup components: Issues, models, and solutions,” in *Web Engineering*, ser. Lecture Notes in Computer Science, vol. 5648/2009. Springer, Jun. 2009, pp. 45–60.

[8] L. Bouillon, J. Vanderdonckt, and K. C. Chow, “Flexible re-engineering of web sites,” in *Proceedings of the 9th international conference on Intelligent user interfaces (IUI ’04)*. ACM, Jan. 2004, pp. 132–139.

[9] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, “A unifying reference framework for multi-target user interfaces,” *Interacting with Computers*, vol. 15, no. 3, pp. 289–308, Jun. 2003.

[10] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera, “A framework for rapid integration of presentation components,” in *Proceedings of the 16th international conference on World Wide Web (WWW ’07)*. ACM, May 2007, pp. 923–932.