Dynamic Mashup Interfaces for Information Systems Using Widgets-as-a-Service

Jesús Vallecillos, Javier Criado, Luis Iribarne, and Nicolás Padilla

Applied Computing Group, University of Almería, Spain
{jesus.vallecillos,javi.criado,luis.iribarne,npadilla}@ual.es

Abstract. Web Information Systems intend to adapt to the users' preferences as new data available on the network. In this regard, the composition and reuse of services which are involved in a web application is an interesting research topic, since these techniques pursue the dynamic construction of applications that can be adapted at design or run time. As for the visualization of these applications, web user interfaces play a key role, serving as a connection point between users and the rest of the system. This article proposes an architecture for specification, storage, management and visualization of components, built from widgets complying with the W3C recommendation, for making web user interfaces. We follow a service-based approach for the interface deployment and communication management, introducing the concept of Widgets-as-a-Service (WaaS). To illustrate this proposal, an example of widget-based Web Information System is shown.

Keywords: GUI, mashup, Widget-as-a-Service, components.

1 Introduction

Web Information Systems are largely dependent on web platform, which is a dynamic and constantly changing domain, that also relies on web services that provide the required data. In this regard, web services are, increasingly, elements that need to be changed and updated in order to be adapted to the new available information and also to the user profiles. Web interfaces do not get out of this necessity and also require the service they offer as an interface to be dynamic and adaptive to the user. With this aim, new projects and proposals have come up in the last few years to build customized web interfaces through the configuration of widgets that the user wants to visualize [8]. For these applications, the user has one or more graphical interfaces available that he/she can configure to create some kind of dashboards. These interfaces are built according to graphical components of high or medium granularity (that is, they are not simple buttons or text fields) that group together some functionalities related to each other and give rise to *mashup* applications based on widgets [3]. Some of the most interesting and currently supported projects are Netvibes (`http://www.netvibes.com`), MyYahoo (`https://my.yahoo.com`) or Ducksboard (`https://ducksboard.com`).

R. Meersman et al. (Eds.): OTM 2014 Workshops, LNCS 8842, pp. 438–447, 2014.

Our research work is focused on dynamic management of component-based graphical user interfaces (GUIs). From our perspective, the representation of this type of interfaces can be "abstracted" in order to create a definition through fragments or pieces analogously to a *bottom-up* approach within the *Component-based Software Engineering* (CBSE). At subsequent steps, we can manipulate this abstract representation to adapt it to the changes according to the user's preferences or any other changes derived or not from the interaction and then make again the graphical interface shown to the user (see Figure 1). This mechanism allows us to work with simplified models of interfaces and later obtain new widgets to be shown as a service dynamically embedded into the GUI (which have been named *Widgets-as-a-Service*, WaaS). This research uses web technologies to facilitate exchange and sharing of information, and also to allow users to cooperate and integrate data.

On the other hand, the components in most widgets-based GUI proposals are isolated from each other, that is, there is no relationship or communication between them. Those proposals that do not deal with isolated components have the difficulty of communication, especially in the web domain, as the communication between some elements may cause some security problems that must be taken into account and solved at design time when programming widgets and application [4]. In this sense, our approach certainly allows relationships and communication between widgets and it suggests an indirect communication mechanism by means of a service that mediates between their components.

As an application scenario for the web domain, this article focuses on widget-based web user interfaces that are being developed for the Environmental Information Network of Andalusia (REDIAM) as a transfer of the research results within the framework of ENIA project (`http://acg.ual.es/enia`), a project

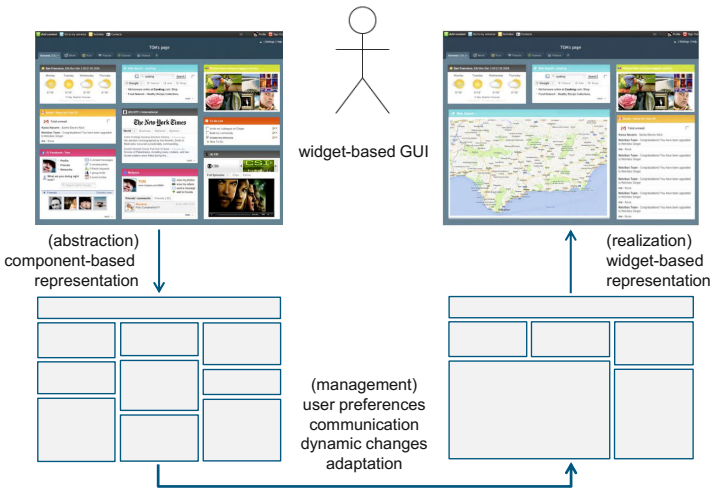

Fig. 1. Our view of widget-based GUI

of excellence funded by Junta de Andalucía (ref. TIC-6114). This project deals with geographic information and users should have a GUI that enables them to access to the multiple services available. For this, each user has access to a graphical interface that allows him/her to manage different widgets that offer certain services such as loading layers of geographic information on a map from OGC services (`http://www.opengeospatial.org`) provided by REDIAM. We also consider these widgets as services to which users can access.

The rest of the paper is organized as follows. Section 2 revises some related work. Section 3 describes the structure of the WaaS model for GUI, in which we analyze the various layers of the architecture developed and the data model to define the components. Section 4 shows a case study. Finally, Section 5 presents the conclusions and future work.

2 Related Work

Next we are going to revise some works focused on the development of *mashup* web applications, architecture systems that manage user interfaces built from components and also some works that study how to establish the communication between components.

In [1], the authors show a system that holds fast and intuitive development of mashup construction. This system is based on a mechanism of autocompletion of graphical user interfaces. The authors propose a method to link the components with each other in order to help the users to complete their workspace in the web application. These links help users to decide which components describe their application. Then, the user selects the components that are going to be loaded into the user interface when the application is deployed. In contrast to our proposal, this approach focuses on RSS components instead of W3C widgets. Furthermore, in [7] the authors perform another approach to carry out a ranking of components considered as interesting by the users. The purpose of this ranking is to make the selection of RSS components easier. Then, the mashup web applications are generated from these components. Both aforementioned works perform an initial load of the components, but it is done at design time. After that, the users can perform their tasks by using the components that have been loaded in their workspace.

On the other hand, in [6] we can observe an architecture proposal where a deployment of web-based components is carried out. The authors developed an architecture to include dynamic applications through a web-based system. They studied the architecture through layers, one of which is in charge of managing the communication between the components and the system core. This form of communication is similar to the one explained in this work. Unlike this work, they did not focus on using the web standard of widgets of the W3C, but rather on the use of Portlet [11] to build the applications.

We can find other works specialized in defining architectures for the management of widgets such as [5]. Here the authors proposed a new type of system for widget-based digital television platforms. They focused on widgets that are

built with web technology rather than widgets that follow the web standard of the W3C. Unlike the means of communication described here, based on a JavaScript server, the authors used a mode of communication via AJAX to obtain the information stored in XML in a server. Another platform where we can use widget-based architectures is mobile applications. In [12] the authors defined an architecture made up of a widget component container that manages the interaction and a software platform that manages its dynamic life cycle. These widgets operated by the architecture are specific for the mobile platform which we are working with.

There exist other works like [13] which make adaptations of the code of the graphical user interface according to the user's preferences. The authors made the GUI more dynamic by developing it according to widget components. Unlike the widget components based on the web standard of the W3C used in this proposal, they implemented these components by making use of Java Swing. Moreover, they defined seven types of roles in order to improve the adaptation and reloading of widgets. Depending on the type of role, there are widgets considered as preferential to make their adaption.

3 WaaS-Based Architecture

In order to offer Widgets-as-a-Service (WaaS), we needed to develop an architectural system to support it. Furthermore, to isolate the proposal from the domain and to be able to manage component-based applications for different platforms, we built the system according to a three-layer architecture. Let us see some features of the layers of this architecture as well as the used data model.

3.1 Architecture Layers

The **client layer** is the upper layer of the architecture (see Figure 2). In our case, the client uses the services that the architecture deploys, coming from the "platform dependent" layer. In our example domain, the client is a web application, which means that this application has been developed under this technology and it must be accessed through a browser. Such application is built on the basis of components that, as it is web technology, have been implemented through widgets. The widgets in which the application has been developed, follow the recommendation of widget of the W3C. Moreover, the application is supported on the services that the architecture deploys to obtain the functionality it requires. By means of these services of the system architecture, the component-based web application can be initialized, receive support for the communication between components and give support to the components that form the application.

The **platform dependent layer** constitutes the intermediate layer of the system architecture. This layer deals with providing the client with the required services and interacting with the independent part of the platform (the bottom level), thus getting some services from it and providing it with others. Regarding the first services, (Figure 2-(a)) this layer gets the code necessary to create the

start-up web application and consults the path to be followed by the information to communicate the components with each other. We achieved this by using web services implemented in the independent part of the platform (Figure 2-(b)). On the other hand, this layer provides the independent part of the platform with some information about the components it manages, such as the direction of the widgets instances which it will use to create the code defined by the web application. In these layers, we can find not only the web client that carries out the functions of the graphical user interface with the system, but also those servers that have been specifically developed to use our proposal within the domain of component-based web user interfaces (Figure 2-(c)).

On the other hand, the components managed by our architecture are black-box components, in which their behavior is hidden and the component is represented by a specification template. This specification describes both the functional part and the extra-functional part (as well as some additional information) of components and this is the only way through which the system can make use of them. Therefore, the only components managed by our system are those defined and built according to this specification, which have also been registered for use. In order our component specification to be valid for any type of components of third parties, we have extended the specification based on COTS (Commercial Off-The-Shelf) components [10]. Our components are called COTSgets [9], this name comes from COTS and gadgets, being understood a gadget as a software artifact that encapsulates a certain functionality. For further details about the structure and content of the component specification used, please see `http://acg.ual.es/definitions/component.xsd`.

Furthermore, we developed the architecture of our system by using a **platform independent layer**. This layer has a server providing the system services that are valid for all platforms (Figure 2-(d)). For this, its functionalities are based only on the description of components and their relationships, regardless of the platform where they will be deployed. It is true that, at the deployment level, there are certain characteristics of each platform that should be taken into

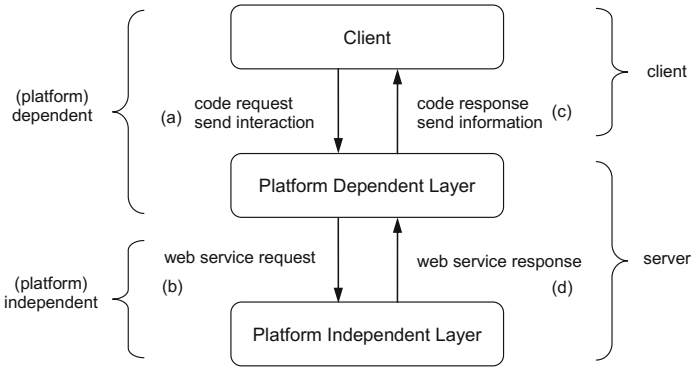

Fig. 2. WaaS-based Architecture Layers

account. For instance, the initialization of a web user interface architecture may be different from any other component-based architecture implemented in Java; or the invocation of methods between components, as part of a communication task, might be different as well. However, there is a common area, an abstract view of such behaviors that can independently be extracted and implemented from the platform. As regards the above examples, we referred to "component architecture initialization" or "communication management", respectively.

Therefore, the services offered in this layer are the same for all platforms, hiding the distinction between different platforms and allowing our proposal to be modular and scalable, and be able to gradually add new functionalities for new platforms supported by the system. As this server has the core of the proposal for the management of component-based architectures, it has been named as *COScore* (COTSget-based architecture Operating Support).

3.2 Component Data Model

Although this article is based on dynamic deployment of widgets in web user interfaces, our ultimate goal is that the system can be able to manage component-based architectures for different platforms. Moreover, another objective is that the components managed by the system should not be just those predefined by the proposal developers but some external developers and third-parties can also take part in the extension and update of repositories. For both reasons, we developed a three-level data model as shown in Figure 3 (described as follows).

The ECR **(External Component Repositories)** level corresponds to the external repositories of components, that is, the repositories of third-parties. These components are stored in their place of origin. This level is dependent on the platform because the components that it stores are specific for the mechanism they are assigned to. The ECR set is formed by each of the external repositories (ER) to be taken into account, that is, $ECR = \{ER_1, ER_2, \ldots, ER_n\}$.

The next level MCR **(Managed Component Repositories)** has a set of repositories of components managed by the system. This level stores the components that meet the structure and the construction guidelines established. It is also dependent on the platform since it is formed by modified versions of the ECR components so that they can be managed by our system and by a set of components developed for the system (MR_0). Therefore, $MCR = \{MR_0, MR_1, MR_2, \ldots, MR_n\}$, being MR_i ($i > 0$) a subset of ER_i for which the components have been embedded with the information and behavior required. At the same time, each MR repository is formed by a set of components (C) and a set of components instances (CI) created and associated to such components. Each instance is created with a piece of information related to the user or client for whom the component was instantiated. This information contains, among other data, the instance identifier, the associated user and the component state. With this information, the system will be able to provide each user with the corresponding set of instances in the proper state.

Finally, the CS **(Component Specifications)** level stores component specifications. These specifications are referenced from the models that the

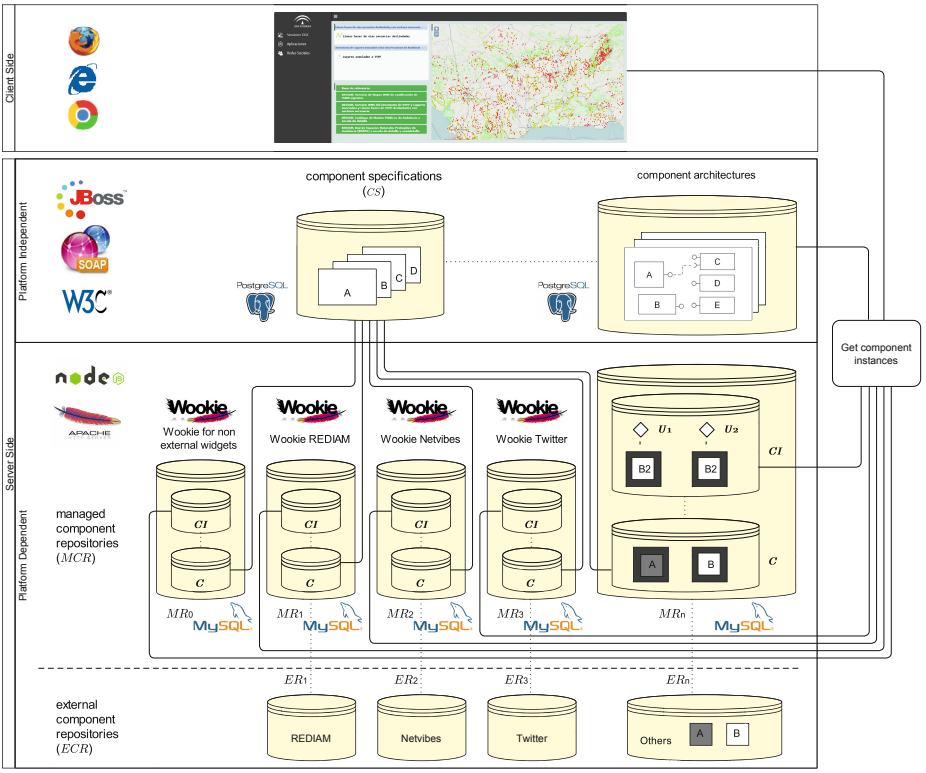

Fig. 3. Component Data Model for widget-based web user interfaces

component architectures describe so as to indicate which component should be chosen for its construction. When a developer registers one component in the system, he/she must register not only the component in the corresponding MR repository but its specification, too. That is the only way through which the system can take into account this component for future architecture construction and so that it can be embedded into a resulting component-based application. These specifications are managed by the platform independent server.

Our proposal is oriented to the deployment of services in the cloud. Therefore, this data model seems to be suitable for the use of cloud technologies, being able to transfer each layer to different servers or even deploy each repository in different servers. In this way, the components, the instances and even the specifications will be considered from the point of view of "resource-as-a-service" (RaaS). In this scenario, the level that is dependent on the platform corresponds to the domain of web applications and consequently the repositories managed by the system (MCR) store widgets. These widgets either have been specifically developed for the system (MR_0) or built from external repositories (ECR). In the second case, two different situations may occur: (a) that the external repositories store widgets, in which case we should create a *wrapper* upon the

existing widget that can perform operations on it in order to be managed by our system; (b) or that the external repositories store a service that is not a widget, in which case we need to create a widget that uses the service concerned.

Figure 3 shows how the system can manage external widgets created from services of REDIAM (`http://www.juntadeandalucia.es/medioambiente/site/rediam`), Netvibes or Twitter (`https://twitter.com/settings/widgets`). In this way, the system has all the elements available to be able to manage the widgets as a service (Widgets-as-a-Service) and dynamically build the web GUIs.

4 Case Study

In this section, we present a case study in order to test the behavior and applicability of the work. This example scenario describes a web user interface that is dynamically constructed from components by applying the proposed architecture layers. The case study has been developed within a project of excellence funded by Junta de Andalucía (ref. TIC-6114), which deals with geographic information. For this purpose, we have built a web application which allows us to load visual layers with geographic information. These layers offer data obtained from an example set of OGC services provided by the REDIAM.

In this case, the OGC services and the rest of operations related to the management of geographic information, are encapsulated into widgets and therefore they are treated as Widget-as-a-Service (WaaS). By deploying these services, the application allows users to view and query geographic information about the soil under study within the project (see Figure 4). The application is made up of three widgets: a map that displays the visualization area and wherein the geographic information is loaded (`Map`); another widget that allows us to select which layers of geographic information will be loaded on the map (`Options`); and a legend

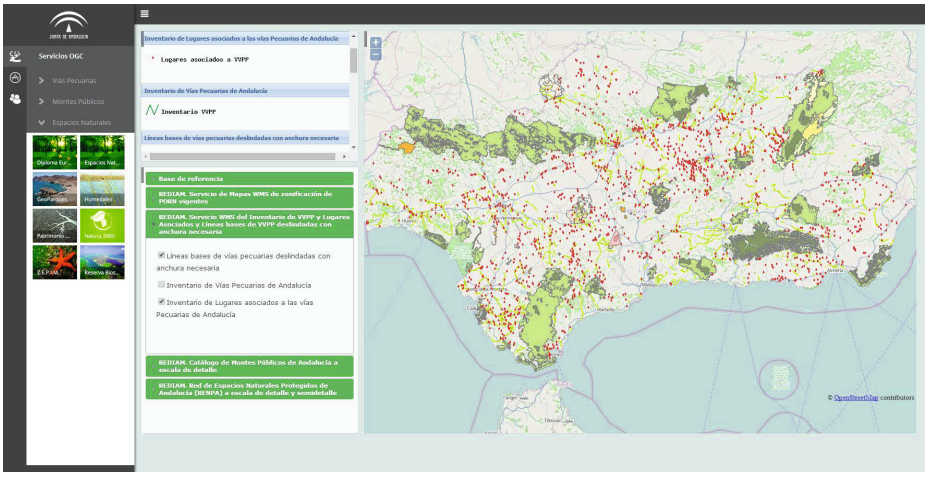

Fig. 4. Widget-based Web User Interface

where information about the layers loaded on the map is shown (`Legend`). This application is available at the address `http://acg.ual.es/enia/uiexample`.

Starting from this scenario, the web application behavior is defined as follows. When a user interacts with the system and accesses to the web address, the application is loaded into the browser. The web page that is accessed does not contain the code for the widgets statically, but the code is constructed dynamically and at run time. To execute this dynamic construction process, JavaScript (Node.js) server performs the role of mediator between the application and the COScore of the architecture. Then, the platform independent server decides which components are going to be loaded and returns the HTML code to Node.js. This code must be embedded in the web application, which contains the components that define the web user interface. Next, this code is sent to the web application, and the client is in charge of inserting the three widgets.

In the following steps, the user interacts with the widgets, and this interaction may involve either exchanging information between these components or notifying the system. In this communication process, the Node.js acts again as a mediator. This server receives the client interactions and invokes the web service of the platform independent layer that is in charge of solving the communication process. Then, within this service, the COScore server resolves the message content and the target widgets and sends the response to the JavaScript server. Finally, the Node.js server sends the message to the corresponding widgets of the user interface. As an example of communication, let us suppose that the user selects a layer in the `Options` widget. This interaction is sent from the widget to Node.js and the message contains information about the source and the selected layer. Then, Node.js invokes the web service of COScore obtaining as a response that this message must be routed to the `Map` and `Legend` widgets. Consequently, the first widget displays the layer into the map, and the second one adds some information about the selected layer to its content.

5 Conclusions and Future Work

In this article we present a three-layer architecture for the specification, storage, management and deployment of component-based applications. Our aim is to abstract the definition and manipulation of such applications to describe behaviors that can be valid in multiple platforms. Within this architecture, the client layer deploys the applications or communicates with the rest of the system functionalities through services offered by the platform dependent layer. In turn, this second layer communicates with the platform independent layer, which offers services like application initialization and management of communication between components. In order to understand this architecture, we also describe the component data model developed within it.

With the aim of validating our approach, we have chosen the web domain as an application platform and we have presented a case study that showed an example developed for a regional research project. In this way, we described the use of widgets as a service (Widget-as-a-Service, WaaS) to dynamically build

component-based GUIs. As future work, we would like to increase the number of application scenarios and validate the proposed architecture in other platforms through practical examples,. We also aim to improve the user's experience when managing widget-based graphical user interfaces [2].

Acknowledgments. This work was funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2013-41576-R, and the Spanish Ministry of Education, Culture and Sport (MECD) under a FPU grant (AP2010-3259), and the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

References

1. Abiteboul, S., Greenshpan, O., Milo, T., Polyzotis, N.: Matchup: Autocompletion for mashups. In: IEEE 25th International Conference on Data Engineering, ICDE 2009, pp. 1479–1482. IEEE (2009)
2. Cechanowicz, J., Gutwin, C.: Augmented Interactions: A Framework for Adding Expressive Power to GUI Widgets. In: Gross, T., Gulliksen, J., Kotzé, P., Oestreicher, L., Palanque, P., Prates, R.O., Winckler, M. (eds.) INTERACT 2009. LNCS, vol. 5726, pp. 878–891. Springer, Heidelberg (2009)
3. Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., Casati, F.: Understanding UI Integration: A survey of problems, technologies, and opportunities. IEEE Internet Computing 11(3), 59–66 (2007)
4. De Keukelaere, F., Bhola, S., Steiner, M., Chari, S., Yoshihama, S.: SMash: secure component model for cross-domain mashups on unmodified browsers. In: 17th WWWW, pp. 535–544. ACM (2008)
5. Fan, K., Tang, S., Liu, Y., Zhang, S., Wang, Y., Xu, Z.: A System Architecture of Widget-Based Digital TV Interactive Platform. In: ICGEC, pp. 360–363. IEEE (2012)
6. Gmelch, O., Pernul, G.: A Generic Architecture for User-Centric Portlet Integration. In: 14th CEC, pp. 70–77. IEEE (2012)
7. Hassan, O., Al-Rousan, T., Taleb, A., Maaita, A.: An efficient and scalable ranking technique for mashups involving RSS data sources. Journal of Network and Computer Applications, pp, 179–190 (2013)
8. Hoyer, V., Fischer, M.: Market overview of enterprise mashup tools. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 708–721. Springer, Heidelberg (2008)
9. Iribarne, L., Criado, J., Padilla, N.: Using COTS-widgets architectures for describing user interfaces of web-based information systems. International Journal of Knowledge Society Research (IJKSR) 2(3), 61–72 (2011)
10. Iribarne, L., Troya, J.M., Vallecillo, A.: A trading service for COTS components. The Computer Journal 47(3), 342–357 (2004)
11. Law, E., Müller, D., Nguyen-Ngoc, A.: Differentiating and Defining Portlets and Widgets: A survey approach. In: MUPPLE 2009, pp. 123–131 (2009)
12. Pierre, D., Marc, D., Philippe, R.: Ubiquitous Widgets: Designing Interactions Architecture for Adaptive Mobile Applications. In: DCOSS 2013, pp. 331–336 (2013)
13. Shirogane, J., Iwata, H., Fukaya, K., Fukazawa, Y.: GUI Change Method according to Roles of Widgets and Change Patterns. IEICE Transactions on Information and Systems 91(4), 907–920 (2008)