A Web Services Infrastructure for the Management of *Mashup* Interfaces

Jesús Vallecillos, Javier Criado, Antonio Jesús Fernández-García(✉), Nicolás Padilla, and Luis Iribarne

Applied Computing Group, University of Almeria, Almeria, Spain
{jesus.vallecillos,javi.criado,ajfernandez,npadilla, luis.iribarne}@ual.es
http://acg.ual.es

Abstract. In the technological world of today, user interfaces (as an essential part of many software applications) are constantly changing in order to meet the needs of different users and adapt to their environment. Accordingly, there is a need for mechanisms to carry out these change processes. This article describes a structure of web services which support the adaptation which constructs mashup type web user interfaces. These interfaces are constructed using third party component architectures, called COTSgets.

Keywords: CBSE · MDE · Web service · Component · Architecture · Mashup

1 Introduction

In today's world it is uncommon for software applications to be static and unchanging. Rather, as is increasingly necessary, applications are adapted, modified and updated over time in response to the demands of users. With user interfaces being an essential part of some software applications, it is necessary that they can also adapt as users change their preferences or modes of use. This has led to new projects and6 proposals in recent years which allow the construction of custom user interfaces through configuration of their interface. In these proposals, the user typically has a Graphical User Interface (GUI) which can be configured to create a bespoke desktop or workspace. The interfaces are made from coarse-grained components (*i.e.* components with fairly complicated functionalities) in order to create *widget*-based *mashup* applications [20]. Examples of these types of interfaces can be found in MyYahoo, Ducksboard or Netvibes [17].

When considering this idea it is often useful to have a software system to manage these user interfaces. Examples of functionalities which can make use of this type of management are: initialising the interface according to either the user's profile or last interaction with the components; saving all the events and actions performed on the interface, or serving as an intermediary in the communication process between components. With these in mind, this article

A. Norta et al. (Eds.): ICSOC 2015 Workshops, LNCS 9586, pp. 64–75, 2016.
DOI: 10.1007/978-3-662-50539-7_6

describes a web service infrastructure which allows the dynamic management of component based user interfaces.

The proposed work uses four principal concepts as foundations. Firstly, the research is applied to the domain of *mashup* user interfaces described by component architectures [11]. Secondly, *Component-Based Software Engineering* (CBSE) [8] techniques are used to construct the user interfaces which allow the applications to be custom built for each user and to change over time. User interfaces are based on third-party components, named as COTSgets (from the combination of COTS and widgets). Thirdly, *Model Driven Engineering* (MDE) [9] techniques are used to produce abstraction mechanisms on the *mashup* interfaces and allow their formal representation. Finally, cloud computing concepts [13,19] are used enabling the component architectures to be managed by web services.

The proposed service infrastructure is based on an architecture with three layers: (a) the client layer, (b) the server side layer which is platform dependent, and (c) the server side layer which is platform independent. The client layer is made up of a user interface constructed from a set of components as described in a component architecture. The platform dependent layer provides the client layer with the services it needs to operate (*e.g.* services related to component communication). It also interacts with the platform independent layer, providing it with some services (*e.g.* services relating to component instantiation) and receiving others (*e.g.* services relating to creating sessions for each user to interact with the interface). The platform independent layer provides a set of services and operations that are common to all possible platforms and can therefore be extended to interfaces other than web *mashup* types. This article will look at the infrastructure which defines the independent layer's set of services. The development of this service infrastructure is focused on the dynamic and flexible management of the component architectures that make up the system. Additionally, the service infrastructure establishes persistence mechanisms for storing and handling these architectures.

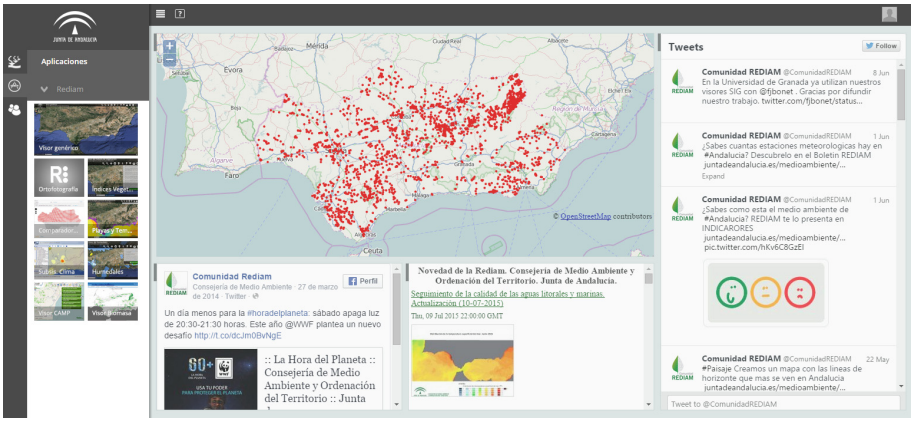

Fig. 1. Web application for ENIA

As an application domain, this service infrastructure has been used to manage a *mashup* user interface of a Geographic Information System (GIS) in the ENIA research project [10], an intelligent Environmental Information Agent. One of the most commonly used data types in these interfaces is obtained from OGC web services developed by the Andalusian Environmental Information Network (REDIAM) [16]. Figure 1 shows an example *mashup* user interface of this GIS [10]. There are a range of services and components on the left side of the interface that can be added to the right side, where the user interacts.

The rest of this article is structured as follows. Section 2 defines the proposed service infrastructure for managing architectures of *mashup* interface components. It then details the three infrastructure levels (databases, drivers and modules) and the public/private services implemented. Section 3 details the implementation of public web services. Section 4 discusses related work, and Sect. 5 presents the conclusions and future work.

2 Multi-service Infrastructure in *mashup* Interfaces

A series of web services, located in the platform independent layer of the Cloud infrastructure, have been created in order to support the component-based architectures of *mashup* interfaces. These services have been organised into two levels according to privacy, see Fig. 2. Public Services are found on the first (highest) level. These include: *Session Web Service*, *Interaction Web Service*, *Communication Web Service* y *Component Web Service*. The Public Services are used to provide functionality, persistence and support to applications which have been built from a component-based architecture [18]. Private Services are found on the second level. These include: *Architectural Model Web Service*, *User Web Service* y *Register Web Service*. These are used to perform certain management tasks such as those related to architectural models, users and the system's available components. Both levels are described in Sect. 2.2.

The *Modules*, *Controllers* and *Databases* levels are found below the two web services levels. The *Modules* level is used by web services and implements all of their functionality. The *Controllers* level manages the different databases which control the environment. Finally, there is the *Databases* level which contains the different databases used to store architecture models, components for user applications, etc. Next, the final three levels are described in more detail.

2.1 Basic Web Services Support

As stated in the previous section, and as can be seen in Fig. 2, there are several levels which support web services. The *Modules* level is the centre of the environment and is responsible for implementing the offered functionality to the user applications by means of web services.

The following modules make up this level: (a) *Lifecycle and Relationships Management Module* (LRMM). This module is responsible for handling the abstract representation of the interface by managing the components and the

relationships between them. It is also responsible for handling the component states; (b) *Display Management Module* (DMM). This module is responsible for handling the component visualisation by adapting the device it is working on; (c) *Transaction Management Module* (TMM). This module allows the exchange of messages and the coordination between components to be controlled. This communication between components is synchronised, *i.e.* all the messages sent by one component are instantly received by the others; (d) *Interaction Management Module* (IMM). This module provides an environment where the user interaction on the interface can be managed. Although the IMM module cannot access user events within the components, the events occurring in the environment (*e.g.* move the component) are saved to learn about user behaviour and adapt the interface; (e) *User Management Module* (UMM). Used for administering users in the environment, processing their registration and modifications in the core of our infrastructure (named as COScore). It can also check to see if a user can log in; (f) *COScore Session Management Module* (COSSessionMM). For each user application running in the environment, the COScore creates the LRMM, IMM, TMM y DMM modules. These modules continue their execution as long as the user session is open.

The modules make use of a series of controllers (*Controllers* level) from the following databases which manage the environment all of them implemented in PostgreSQL: (a) *Architectural Models and Users*. This database stores both the applications' architecture models and the environment's users. It is handled by the *ManageArchitecture* controller by means of the mapping framework Hibernate (http://hibernate.org/) which manages the interface's architecture models as objects; (b) *Interaction*. This database saves the interactions which

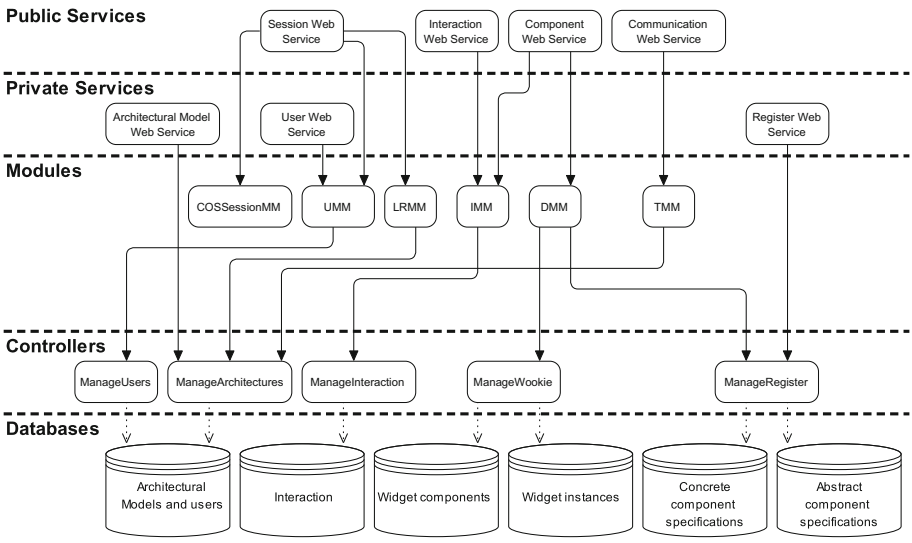

Fig. 2. Web services infrastructure (COScore)

occur in the environment, such as; adding and removing components, changes in size or position as well as communication processes; (c) *Widget components*. Stores all the Widget components which may be needed by the Web applications. These components then generate the instances which are embedded in the user interface. This repository is supported by a server of widget components called Wookie (http://wookie.apache.org/), which follows the W3C standard [15]; (d) *Widget instances*. This repository stores the instances of Widgets that are associated with each user's Web application; (e) *Concrete component specifications*; (f) *Abstract component specifications*. Registers the abstract component specifications, independent of the platform.

2.2 Private and Public Services

As previously stated, the services provided by COScore are organised in two levels. The first level contains the public services, which can be directly used by the user applications; the second level contains the private services, which perform certain management tasks within the COScore, but cannot be accessed by applications.

The **private services** are: *Architectural Model Web Service*, *Register Web Service* and *User Web Service* (Fig. 2). The purpose of the *Architectural Model Web Service* is to handle the component architectures used to describe the *mashup* user interfaces. The system uses two different architecture models to manage these architectures, the abstract architecture model and the concrete architecture model. Using *Model Driven Architecture* (MDA) as a basis, the architecture models are used to define of the user interface at different levels of abstraction. The abstract architecture models allow a platform independent user interface to be defined in terms of the existing types of interface components and the relationships that exist between these components. These models correspond to the *Platform Independent Model* (PIM) in MDA. Furthermore, concrete architectural models allow a user interface to be defined based on the concrete components used in a given platform. Likewise, these architecture models correspond to the *Platform Specification Model* (PSM) in MDA. The web service allows us to add or remove an abstract or concrete model in order to manage these models. Another private web service is the *Register Web Service*, which allows abstract and concrete components to be registered and removed in the system. Finally, there is the *User Web Service*. This service manages the users and carries out basic functions such as adding and removing users, and checking and modifying users' information.

The **public services** (*Session Web Service*, *Component Web Service*, *Communication Web Service* and *Interaction Web Service*) support the *mashup* user interfaces. *Session Web Service* manages user sessions in the environment. One of its tasks is to check whether a user belongs to the system (*Login* operation), initializing the modules for that user. Another of the tasks carried out by this service is the initialization of the user interface. This task reads the component model, generates routing tables, creates component instances and returns the user interface code which has been generated. Finally, by means of the *Logout*

operation, this service allows the session to be closed and eliminates the components pertaining to the user. The *Component Web Service* manages the handling of the components in the user interface, i.e., adding and removing components. The *Communication Web Service* manages the communication between components. It receives a message from a component and gets which other components the information should be sent on to. The *Interaction Web Service* is responsible for storing information on how the user interacts with the application. This interaction relates to changes in component position and component size, adding and removing components to the user interface and registering the communication processes between components.

3 Implementation of Public Web Services

This section will explain some operations connected with the public services found in the proposed structure. To describe these services, Business Process Model Notation (BPMN) diagrams will be used to show their operation and the flow of information which takes place. Figure 3 shows the *Login* and *Init User Architecture* operations of the *Session Web Service*.

As described below, both operations are related. The *Login* operation also involves executing the initialization operation. When the *Web application* location is accessed from the browser, the first action is obtaining the HTML code of that application (by using *Request web application*). Next, the user is logged in. This requires the application to communicate with the *JavaScript Server* by using *Process Login request*. Subsequently, the JavaScript server invokes the *Login* operation of *Session Web Service*. This operation makes use of tasks involving the UMM and COSSessionMM modules. Once it has been checked that the user is registered in the system, the user ID is sent to the application. During this process the *Login* response function (in the *JavaScript Server*) invokes the web service's *Init User Architecture* operation. As this initialization interface is executed, the model associated with the user is read (by *Read concrete architectural model*), and the structure for communication between the architecture components is generated (by *Generate routing structure*). The component instances are then created (by *Generate web concrete components instances*) and the necessary code to generate the user interface is returned (by *Generate code for user instances*). The modules involved in this initialization process are TMM, LRMM and DMM. Once the code has been sent to the Web application, the widget instances that have been created (and located in the Wookie server) are embedded in the web page.

Figure 4 shows another example operation (*Add component*). This belongs to the *Component Web Service*. The BPMN diagram shows that the request to add a component to the user interface comes from the *Web application*. The requests are first processed by the *JavaScript Server*. This invokes the operations which correspond to *Component Web Service*. Subsequently, the DMM module executes the task responsible for adding components to the architecture (*Process to add component*). For this, the module communicates with the *ManageArchitectures* controller, which accesses the *Architectural model* database to

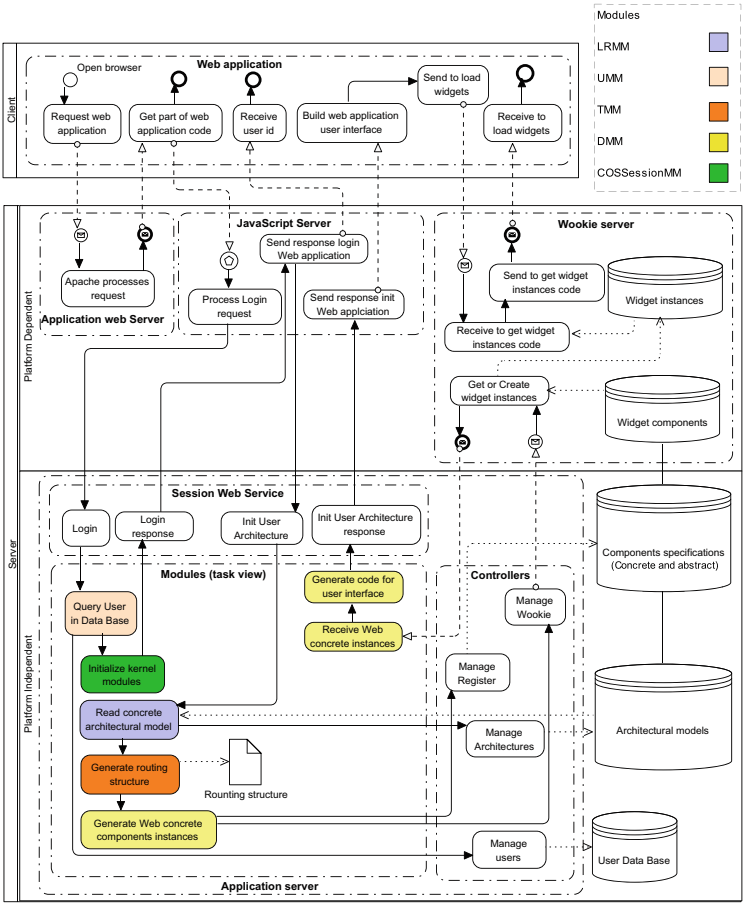

Fig. 3. *Login* and *Init User Architecture* operations of *Session Web Service*

modify the corresponding Architectural model. The operation of adding components involves storing the interaction with the interface requiring the use of the *Interaction Management Module*. By way of *Store Interaction*, this module stores this interaction using the controller *ManageInteraction*. Once these operations have been done, the task *Process to add component* responds by sending the component to be added to the user interface.

In order to demonstrate how the functionality of the web services has been developed, the following example shows the implementation details. The operation *Add component* of the *Component Web Service* is described. Figure 4 shows that for this operation *Add component* is executed first (task 1). Since the user interface is a Web application, this task is implemented using the JavaScript language. Task 1 of Table 1, lines 3–5 show the code which corresponds to the beginning of this operation. Subsequently, the information is sent to the JavaScript

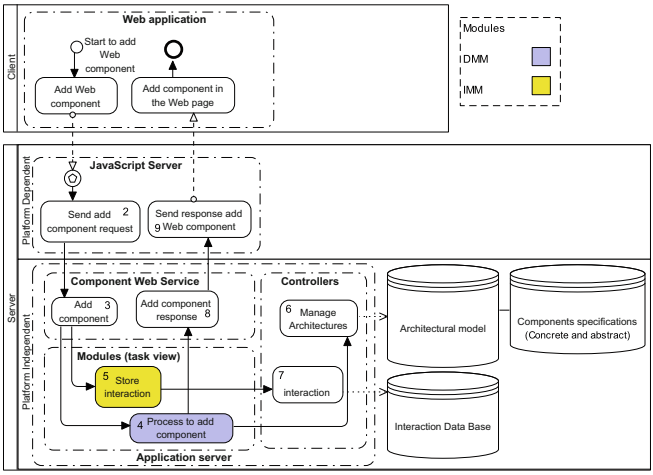

Fig. 4. *Add Component* operation of *Component Web Service*

Table 1. A piece of the code of the Add component operation

tasks	code

1, 10	`<html> <head>`
	` <script> ...`
	`  function addComponent(componentId) {`
	`  websocket.emit('addComponent', {userID: uid, componentId: componentId}); }`
	`  websocket.on('addComponent', function(data) { $('\#main').append(data);});`
	`  ...`
	` </script> </head> ...`
	`</html>`

2, 9	`socket.on('addComponent', function(data) {`
	`  var argsAddComponent = {userID : data.userID, componentId : data.componentId};`
	`  callWS('http://...', 'addComponent', argsAddComponent, function(wsResponse) {`
	`  wsResponse.forEach(function(value, index) {`
	`  io.sockets.in(data.userID).emit('addComponent', value.codeHTML); }); });`
	`});`

3, 8	`public class COSWSImpl implements COSWS { ...`
	`  public List<ComponentData> addComponent(String userID, String componentId,`
	`  String componentName) {`
	`  List<ComponentData> result = null;`
	`  Context initialContext = new InitialContext();`
	`  COSSessionMM cossmng = (COSSessionMM)initialContext.lookup("...");`
	`  DMM dmm = (cossmng.getUserEJB(userID)).getDMMS().get(0);`
	`  IMM imm = (cossmng.getUserEJB(userID)).getIMM();`
	`  result = dmm.addComponent(componentId);`
	`  imm.setCAM(dmm.getCAM());`
	`  imm.loadInteraction(dmm.getCAM().getCamID(), userID, componentId,`
	`  componentName, 'addComponent');`
	`  return result;`
	`  } ...`
	`}`

4	`public class DMM { ...`
	`  public ConcreteComponent addComponent(String componentId) {`
	`  ComponentComponent result = null;`
	`  ManageArchitectures ma = new ManageArchitectures();`
	`  result = ma.addComponent(componentId);`
	`  return result;`
	`  } ...`
	`}`

5	`public class IMM { ...`
	`  public boolean registerInteraction(String modelId, String componentId,String userId,`
	`  String interactionMoment, String action, String property, String value) {`
	`  ManageInteraction mi = new ManageInteraction();`
	`  boolean insert = true;`
	`  insert = mi.store(modelId,componentId,userId,interactionMoment,action,property,value);`
	`  return insert;`
	`  } ...`
	`}`

server, which acts as a mediator between the client and the platform independent layer. The JavaScript server invokes the corresponding web service using *Send add component request* (task 2 of Fig. 4) by means of the code shown in task 2 of Table 1.

The response to the client application containing the code of the component is also included in task 9 of Table 1, line 4. This is carried out by the task *Send response add web component* (task 9). The information is then received by the web service. The web service and its modules are implemented with Java. The code of the *Add component* task (task 3) is shown in Table 1, "tasks #3 #8". Once the task information has been received, *Process to add component* (task 4) and *Store interaction* (task 5) are invoked. These tasks belong to the IMM y DMM modules. The method shown in the task 8 is returned by the *Add component* operation.

The implementation of the task which stores the interaction (*Store interaction*) is detailed in task 5. It invokes the *registerInteraction* method of the IMM module, which then executes tasks to save the interaction. Subsequently the interaction is stored in the *Interaction Data Base* repository by means of the *Manage Interaction* task. We can see the implementation of the *Process to add component* task 4, which is used to add components. Later, the state of the user architecture is saved in the *ArchitecturalModel* database using the *Manage Architectures* task (task 6). Finally the component to be embedded in the user interface is returned by the *Process to add component* task using the return method shown in task 4.

4 Related Work

By using *mashup* user interfaces it is possible to carry out modifications to user interfaces, adapting them to the user's needs. The project OMELETTE [4] is an example based on the use of *mashup* application technologies to allow users to create their own collaboration platforms. This is achieved by providing a set of tools and components (based on W3C widgets) that support the development of telco *mashup*. They also make use of models to manage the user workspaces. The project differs with regard to this article in that in our environment the focus is on the development of individual applications and desktops are not shared for collaborative tasks. Furthermore, in OMELETTE components use Apache Rave to communicate, which restricts the possibility of communication processes between components other than widgets. In our case, we have used a JavaScript server to interconnect different types of components using Web Sockets. DashMash [3] and ServFace [14] are other examples of similar projects.

A framework is proposed in [5] which allows users to build component based *mashup* interfaces (widget type) to suit their needs. As in our case, they make use of MDE to represent the environment although they are more focussed on supporting web platforms. Our proposal supports multiple platforms.

In [1] an environment called *NaturalMash* is created which allows the construction of *mashup* user interfaces by means of widget type components chosen from a pallet. These components can be dragged and dropped in the

workspace allowing users to design their own manipulation environment. The system includes a way to select implemented components using natural language. Nevertheless, the proposal has some limitations such as the possibility for the components forming the environment to communicate. This limits the interoperability between the components.

With respect to using web services to handle *mashup* user interfaces, there are works such as [2] where they are used to provide applications with the opportunity to share workspaces built with *mashup* user interfaces on different devices. Different model types are used to carry out this process of sharing workspaces. In the process, one model is used to compose the user interface patterns, defining the components that form the workspace. Another model describes the current state of the user interface and another visual template model integrates the representative data with graphic elements. These models are provided by web services using *mashup* applications. In contrast to the work in this paper, a hierarchy of services to control the environment is not performed nor are the services intended to handle any elements other than the models i.e. interaction processes or user management.

Other works such as [12] exist where the focus is on the use of components and *mashup* user interfaces to construct applications appropriate to the user's needs. This is done by users creating their own environments from a collection of components. Service Oriented Architectures (SOA) are used for processing the user environment adaption and communicating between the widgets. As such, they use services to control the user interface management processes and the communication between components in a similar way as in this paper although the proposal is not focussed on handling anything other than web type environments. As such, the ability to apply them to other applications is limited.

5 Conclusions and Future Work

This work describes a web services structure, which has been implemented to offer consistency and functionality to architectures, based on COTSgets components, of *mashup* interfaces. This has been achieved using *Component-based Software Engineering* (CBSE), *Model-Driven Engineering* (MDE) and *Cloud computing*. This structure has been organised into two levels of services (public and private) which use a combination of models and controllers to implement all the functionalities and access the database.

As future work, the environment could be extended to support more client applications based on other types of components (*e.g.* built with Java). This would require the creation of new types of component repositories and a different approach to creating the user interface during the login process. Furthermore, dedicated services could be added to the transformation [7] and regeneration [6] processes. The transformation process could be used to allow the architectures to change at an abstract level. By combining these changes with a regeneration process, real components stored in the component repositories, could be associated with abstract components of the abstract model produced as a result of the transformation process.

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. This work was also supported by the CEiA3 and CEIMAR.

References

1. Aghaee, S., Pautasso, C., De Angeli, A.: Natural end-user development of web mashups. In: Proceedings of the 2013 IEEE Symposium on Visual Languages and Human Centric Computing, pp. 111–118. IEEE (2013)
2. Ardito, C., Bottoni, P., Costabile, M.F., Desolda, G., Matera, M., Picozzi, M.: Creation and use of service-based distributed interactive workspaces. J. Vis. Lang. Comput. **25**(6), 717–726 (2014)
3. Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D., Francalanci, C.: DashMash: a mashup environment for end user development. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) ICWE 2011. LNCS, vol. 6757, pp. 152–166. Springer, Heidelberg (2011)
4. Chudnovskyy, O., Nestler, T., Gaedke, M., Daniel, F., Fernández-Villamor, J.I., Chepegin, V., Fornas, J.A., Wilson, S., Kögler, C., Chang, H.: End-user-oriented telco mashups: the omelette approach. In: Proceedings of the 21st International Conference Companion on World Wide Web, pp. 235–238. ACM (2012)
5. Cinzia, C., Maristella, M., Matteo, P.: A UI-centric approach for the end-user development of multidevice mashups. ACM Trans. Web **9**(3), 11–40 (2015)
6. Criado, J., Iribarne, L., Padilla, N.: Resolving platform specific models at runtime using an MDE-based trading approach. In: Demey, Y.T., Panetto, H. (eds.) OTM 2013 Workshops 2013. LNCS, vol. 8186, pp. 274–283. Springer, Heidelberg (2013)
7. Criado, J., Rodríguez-Gracia, D., Iribarne, L., Padilla, N.: Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces. Practice and Experience. Wiley Online Library, Software (2014)
8. Crnkovic, I., Larsson, M.: Challenges of component-based development. J. Syst. Softw. **61**(3), 201–2012 (2002)
9. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.: A classification framework for software component models. IEEE Trans. Softw. Eng. **37**(5), 593–615 (2011)
10. ENIA Project: Environmental Information Agent. Applied Computing Group, Ref.P10-TIC-6114, Junta Andalucia (2015). http://acg.ual.es/enia
11. Florian, D., Maristella, M.: Mashups: Concepts, Models and Architectures. Springer, New York (2014)
12. Hoyer, V., Gilles, F., Janner, T., Stanoevska-Slabeva, K.: SAP research rooftop marketplace: putting a face on service-oriented architectures. In: IEEE World Conference on Services-I, pp. 107–114. IEEE (2009)
13. Lee, C.A.: A perspective on scientific cloud computing. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 451–459. ACM (2010)
14. Nestler, T., Feldmann, M., Hübsch, G., Preußner, A., Jugel, U.: The ServFace builder - a WYSIWYG approach for building service-based applications. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 498–501. Springer, Heidelberg (2010)
15. W3C: Widgets family of specications. Web Application Working Group, Technical Report, W3C (2012). http://www.w3.org/2008/webapps/wiki/WidgetSpecs

16. REDIAM: Andalusian Environmental Information Network. (2015). http://www.juntadeandalucia.es/medioambiente/site/rediam
17. Sire, S., Bogdanov, E., Palmér, M., Gillet, D.: Towards collaborative portable web spaces. In: 4th European Conference on Technology Enhanced Learning (EC-TEL), Workshop on Mash-Up Personal Learning Environments (MUPPLE 2009) (2009)
18. Vallecillos, J., Criado, J., Padilla, N., Iribarne, L.: A component-based user interface approach for Smart TV. In: 9th International Conference on Software Engineering and Applications. (ICSOFT-EA), 29-31 Aug 2014, pp. 455–463, IEEE. (2014)
19. Whaiduzzaman, Md., Haque, M.N., Rejaul K.C., Gani, A.: A study on strategic provisioning of cloud computing services. Sci. World J. **2014**, 16 (2014)
20. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. IEEE Internet Comput. **12**(5), 44–52 (2008)