A Component-based User Interface Approach for Smart TV

Jesús Vallecillos, Javier Criado, Nicolás Padilla and Luis Iribarne
Applied Computing Group, University of Almeria, Almeria, Spain

Keywords: Component-based Architecture, SmartTV, Services.

Abstract: The fast growth and diversity of technological devices currently being produced is benefiting areas such as “ambient intelligence”. This area attempts to integrate information technology in any personal environment. However, to construct service/application software that adapts to different environments, there must be techniques available that favor this type of development. *Component-based software Engineering* (CBSE) is a discipline of the software engineering that integrates (previously constructed) components to build new software systems. This paper presents a CBSE approach to build *Graphical User Interfaces* (GUI) at run-time. Both a component-based perspective of the user interface and a set of component relationships are presented in the paper. As a case study, this paper also describes an application built for an emerging computation environment, Smart TV. A running example is also presented through the paper putting some Web-based solutions to build User Interfaces together (e.g., Wookie, W3C Widgets, Node.js).

1 INTRODUCTION

We are currently witnessing very fast growth of devices in a diversity of technologies (e.g., smart-device integration at home, such as SmartTV, smartphones, tablets, etc.). One area benefited by the appearance of these new devices is “Ambient Intelligence” (Remagnino et al., 2005). This area attempts to integrate information technology in any personal environment, facilitating daily activities as transparently as possible with respect to the information systems. However, before such integration can be possible, the service/application software must be adapted to the different environments. Therefore, it is very important to have systems that facilitate human-computer interaction in many environments and enable their design to be adapted to this diversity of devices.

Component-based Software Engineering, CBSE (Crnkovic and Larsson, 2001) is a software engineering discipline that can assist in facilitating this interaction and enable the adaptation of many devices during software development. CBSE improves software development by reusing it, contributing reliability, and reducing the time required for creating such software. Contrary to traditional software development, CBSE is focused on integrating previously constructed software components in the construction of the system following a *bottom-up* development perspective instead of a traditional *top-down* one. This concept of reuse and management of components is also present in standards such as IEC/PAS 62814 (Belli, 2013).

As application domain, the research work on *user-interface development* is currently involved on improving new CBSE solutions. For instance, Figure 1 shows an example of one user interface constructed by assembling components that may be seen in the Netvibes-type interface[1]. This kind of user interface (i.e., based on components) gives a wide catalog of components that can be added or eliminated to customize the appearance to the user-interface as well as the services it provides. However, Netvibes components are isolated in the user interface, and therefore, no exchange information among component, limiting possibilities they might offer. This leads to the question of how to create user interfaces based on interrelated components adapted to many environments.

Our research attempts to find an answer to this issue focusing on the development of component-based architectures. In addition, we try these architectures to be deployed or executed in different platforms. Furthermore, it is necessary that the components of the architectures can be interrelated with each other. Once these architectures are deployed, they should not remain static, but they must be able to change and adapt their component structure at run-time. In order to accomplish these **goals**, we have proposed a methodology for adapting component-based architec-

[1]Netvibes web page: http://www.netvibes.com

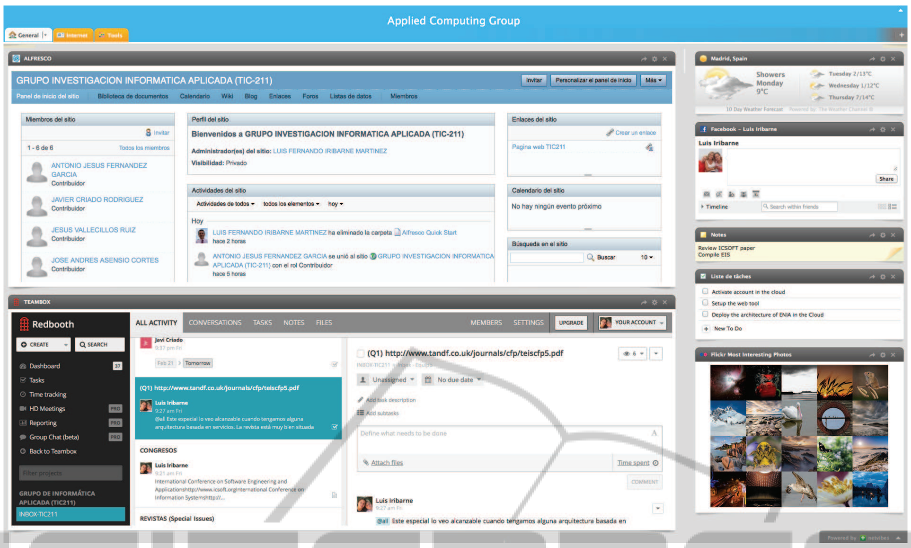


Figure 1: An example of Component-based User Interface.

tures at run-time (Iribarne et al., 2010; Iribarne et al., 2011). This methodology relies on *Model-Driven Architecture* (MDA) levels to describe the component-based software and on a client-server model. Then, components and architectures are defined in two levels of abstraction: *Platform Independent Model* (PIM) level and *Platform Specific Model* (PSM) level. The second one will be used to deploy the architecture on the client side. Therefore, from the PIM level, it could be possible to realize different PSM architectures depending on the platform, by using a regeneration process (Criado et al., 2013). Moreover, this methodology also allows the interaction captured from the client side to modify the architectural definitions. On the other hand, this interaction could only generate the communication between components. Anyway, both processes will be managed in the server side of our system, as a gateway.

The proposed methodology is not suitable for all component-based architectures. It is valid for architectures built from medium/high grained components which encapsulate some independent behavior, but they should or must be able to interact with other components of the architecture. In addition, these architectures should be able to change at run-time with the aim of adapting to the new user's needs or the new system's requirements. One specific domain for the application of such architectures is the development of interactive systems. This article focuses on a component-based interactive system for one Smart TV user interface which was tested in a Samsung Smart TV Emulator[2]. The components used in this environment, included all the necessary for this purpose, such as visual and functional features, are explained. A set of relationships necessary for the communication and adaptation of the component-based architecture was also fixed. However, the management at the PIM level and the realization of the PSM architectures are not addressed, which are issues out of the scope of this paper. Therefore, we can summarize the **contributions** of this paper as follows:

- Application of our methodology in component-based user interfaces for Smart TV.
- Description of components and relationships used in the methodology.
- Implementation of the approach from the point of view of the client and the server.
- Illustration of our approach through a running example.
- Development of a prototype of interactive system, implementing the proposed methodology and available on the web.

The remainder of the paper is organized as follows. Section 2 describes a sort of component-based GUI developed for an interactive Smart TV system. It serves as a running example used through the paper. Section 3 presents some component and relationship issues and it exemplifies them by using the previous GUI. Section 4 explains the technological solution adopted to implement our architecture in the Smart TV environment. Section 5 reviews and discusses some related work. Finally, some conclusions and future work are presented in Section 6.

[2]https://www.samsungdforum.com/Devtools

2 COMPONENT-BASED UI ON SMART TV

The main research of the paper concentrates on developing component-based software architectures for *interactive systems*. Henceforth, we will use CBA to refer to *Component-Based Architectures*. Although a CBA can be applied in several environments, they are currently used to build interactive systems. The user interface in our interactive system (a CBA instance), as shown in Figure 2, could be visualized on different platforms supporting different technologies.

Of course, depending on the technology used in each environment, the interface's functional and graphic features may vary in an attempt to adapt to it. These variations may be necessary for several different reasons; for example, one might be such as the characteristics of the used device (screen size, network connection bandwidth, etc.). To achieve this adaptation, the system has components with similar functionality, but specific to each device. This article does not describe how this adaptation to the environment is done. On the contrary, the goal pursued is to concentrate on a concrete technology, the Web technology, in a concrete environment as defined by Samsung for its Smart TV series. Below, we describe the application developed for this environment, which will serve as the basis for describing all the elements that make up our component-based software architecture and the technology used to put it into service.

The development of applications for Smart TV platforms is booming, and especially applications developed for Samsung Smart TV. Figure 3-(a) shows a CBA application being developed for the Andalusia Regional Government (Spain). This application implements an environmental management system called ENIA[3] based on user profiles. In this application, the users interact for land uses and plant covers information in the Region of Andalusia. In this example, the interface is made up of a set of components associated with the workspace of a technical user profile. Since the Samsung Smart TV platform uses a Web-based technology, the application can easily be adapted to other environments that use the same technology. On the other hand, Figure 3-(b) shows an abstract representation of this interface. This representation makes it easier to understand the functioning of our component architecture, since it enables it to be abstracted from the visual features and working only with its characteristics. Based on this abstract representation, an abstract model of components can be created. This abstraction allows us to describe the component distribution and composition in the interface, leading to a component architecture model.

Let us examine both representations in Figure 3. A `Header` component is located at the top in both figures. This component controls the access to the system and the language preferences. For this task, the `Header` component consists of three components:

(a) the `User` component, which is used to identify the user that interacts with the system,

(b) the `Languages` component, responsible for changing the language of the interface components, and finally,

(c) the `Logout` component, which closes the session of the connected user.

All these components share a common purpose, managing the user profile. Component `Header` is a clear example of how CBSE is used to create more complex components by integrating more basic ones. Other components, used by the user to perform the application domain tasks, appear in the interface. The `Map` component shows a map of the study area with the results of the queries mentioned below. The map was implemented using OpenLayers[4]. Within this component is the `Options` component, which contains the queries that can be shown on the map. At the left of the `Map` component is the `Elements` component, which enables the user to select the components that can be shown on our interface prototype. Finally, underneath `Map` component, are two components (the `Histogram` and `Pie Chart` components) used to show information related to the current query. More information about this running example is available on the web[5].

Having described the application developed, let us see the internal structure of a component and the main interrelationships that among them.

[3]ENIA project: http://acg.ual.es/enia

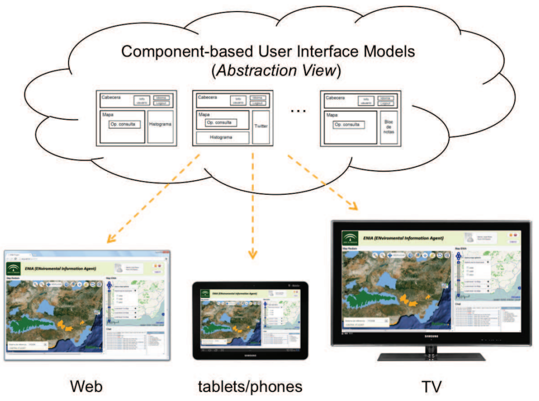
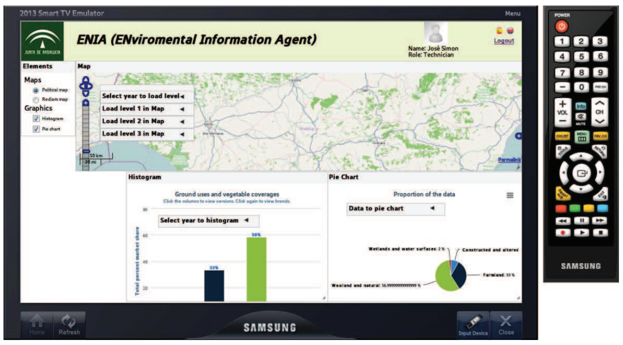


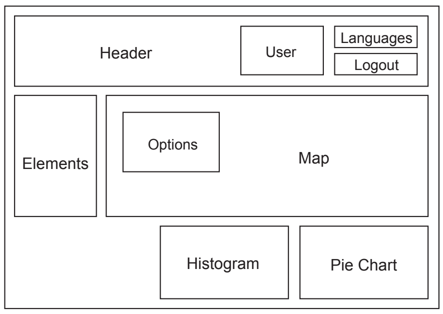
Figure 2: Adaptation of the UI to different platforms.

[4]OpenLayers: http://www.openlayers.org/

[5]Running example: http://acg.ual.es/enia/cbuismarttv



(a) CBA in Samsung Smart TV.



(b) CBA abstract representation.

Figure 3: An interactive system example for Smart TV.

3 COMPONENTS AND RELATIONSHIPS

As observed in Figure 3, a user interface in our system is made up of a set of components. Each component in this interface is divided internally into two modules, the **User Interaction Module**, which manages interaction with the user (and includes the visual feature, when the interface is graphical, such as buttons, checkbox, etc.), and the **Functional Module**, which performs the main tasks of the component (such as accessing the databases, etc.). Each component also has a set of internal **ports** by which it relates to other components, enabling message exchange among them and providing more dynamic, adaptable behavior. Management of these communications is done by a series of functions included in each component. Most components created in an application usually include both modules. However, components can be implemented using only one or none of those modules. Thus three component subtypes may be distinguished:

(a) **Functional Component**. A component including only this module is used basically to implement the underlying functionality of an application. For instance, a component for registering a new user in the system.

(b) **User Interaction Component**. A component that includes only this module basically manages user interaction and shows visual content (when user interfaces are visual, naturally) related to the domain information. For instance, the `Histogram` component used to display some data from the `Map` component.

(c) **Container Component**. A component that does not include any of the mentioned modules, is named as a container. This component subtype is used to contain other components which develop a common task or purpose together. For instance, the header section at the user interface.

All components in a user interface is contained in one special container component called the `Main Container`, which represents the architecture of the user interface, that includes all the components and their relations. Figure 4 shows the inner architecture of the study application, that is, a user interface made up of eight components: `User`, `Languages`, `Logout`, `Histogram`, `Map`, `Pie Chart`, `Elements` and `Options` components.

Moreover, in the system exists a `Kernel` service with three main capabilities, in charge of managing the user interface. The *Lifecycle and Relationships Management* capability manages the life cycle of components along their relationships. *Transaction Management* controls all of the messaging between components through their relationships. Finally, the *Display Management* capability manages and maintains the distribution, design and dimension of the interface components.

Let us go now to analyze the relationships among the application components. A Kernel's *Transaction Management* capability is used for managing communications established among components by the type of relationships existing among them. Each component in the application has a sort of ports used for sending/receiving information to/from another component. To be able to understand the functioning of a port, the concept of a binding relationship must first be defined. A **binding** relationship is a connection between the ports of two components.

There can only be one binding relationship between two components. These binding relationships provide many possibilities by adding performance and restrictions to information exchange.

The **relationships** used in the application are: (a) *composition* ($\sqsupset$); (b) *association* ($\approx$); (c) *dependency*

(≫); and (d) *producer-consumer* relationship (⊢).

A *composition* relationship (i.e., ⊐) shows that a component is included in another and cannot be accessed through its ports by any other component outside the composition. This relationship occurs when the components that form part of the composition have a purpose or target in common, and develop this purpose together. Re-examining Figure 4, we can see that there is a composition relationship between the `Header` component and the `User`, `Logout` and `Languages` components. This relationship exists because they share a common purpose which is to control interface access and provide a language appropriate to the interface components, and they develop this purpose together.

The *association* relationship (i.e., ≈) between two components appears when information necessary to both is exchanged between them. The `Languages` component has an association relationship with the `User` component because the user can change the language of all the components, including the `User` component. Since a component included in a composition cannot be accessed by another component nor in the same composition, the `Header` component has to perform the `Languages` component task. Therefore, there are some association relationships that start out from the `Header` component to the `Histogram`, `Pie Chart`, `Map` and `Elements` components.

A *dependency* relationship (i.e., ≫) exists when a component cannot exist, or its existence makes no sense without another component. Thus, there is a dependency relationship between the `User` component and the `Logout` component because it would make no

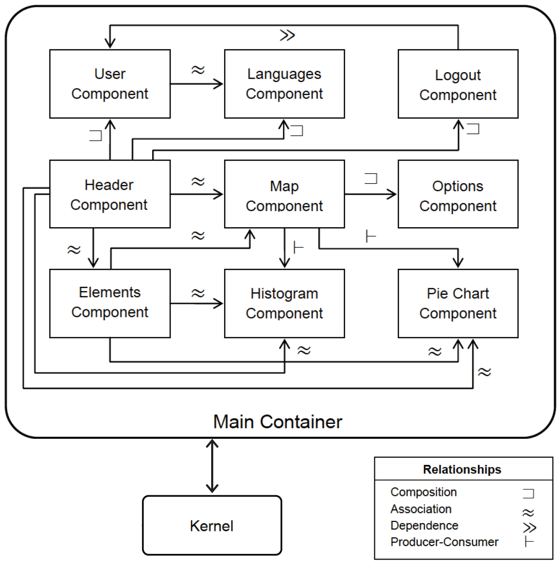


Figure 4: Components and their relationships.

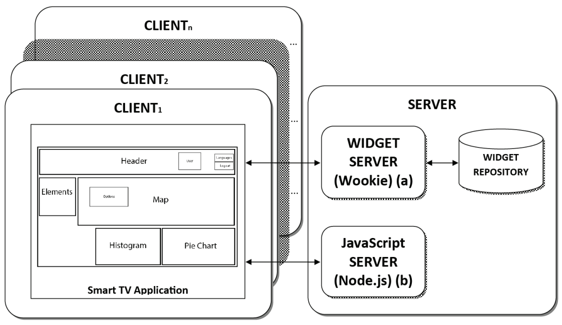


Figure 5: Client-Server Architecture

sense for there to be a `Logout` component in the interface if the `User` component were not there.

Finally, the *producer-consumer* relationship (i.e., ⊢) occurs when a component produces information which is consumed (that is, used) by another component. The `Map` component has two producer-consumer relationships with the `Pie Chart` and `Histogram` components. These relationships are due to the `Map` component, providing information through the selected options and marked areas, useful for components that use statistical graphics.

4 IMPLEMENTATION ISSUES

As mentioned in Section 2, there are Smart TV platforms where applications based on Web technologies are being developed to achieve greater integration with external devices and services. Our component-based interactive system model uses a *client-server architecture*, as shown in Figure 5. The user interface for each user of the system is in the client side. Each interface may be different depending on the user profile and his personal preferences. In the server side of our architecture, we can found a set of servers based on Web technologies. This servers are in charge of managing the component repository and the communication between components. The employed technologies are analyzed in more detail below.

4.1 Client Side technology

As mentioned above, the applications in Samsung Smart TV are based on Web technology. Therefore, our components were designed following this technology using the W3C Widget specification[6]. This specification proposes a structure that is easily adaptable to our component model.

Table 1 shows an example of the internal structure of the `Map` component. There are two main files in the

[6]W3C Widget: http://www.w3.org/TR/widgets/

structure, `index.html` and `config.xml`. These files start up the resources necessary to execute the component. In addition to these two files, the specification proposes a folder structure that contains the functionality of the components and what they look like. In the content folder, there are two subfolders. One is the `interaction` folder, where elements comprising the component interface and the management of events generated by these elements reside. In the example, this functionality is found in the `UIscript.js` file. Apart from this, the `functional` folder also includes the rest of the component functionality. In our example, the file `Taskscript.js` includes the underlying functionality for map management.

A piece of the `Taskscript.js` file is shown in Table 2. It shows the function necessary to receive a map layer (`receiveLoadLayer`) and the function for deleting a map layer (`receiveDeleteLayer`). These functions make up the main functionality of the `Map` component. These two functions are analyzed in detail. It may be seen that the `Map` component communicates with another component, `Options`, through ports by means of the `websocket.on()` function. The first parameter of the function (`receiveLoadLayer`) shows the port of the `Map` component, by which the external information is received. The second parameter is a function which sends the request to the destination component, represented by the `componentName` label. This function represents the behavior to be executed when the port of this component is invoked. This function also has a parameter to describe the name of the layer (`nameLayer`), and a parameter to locate the data to be acquired (`dataLayer`). After this process, the received information must be checked and validated from that expected by the component (if clause). This condition is satisfied because, when the components are loaded in the interface, they are given a unique name which identifies them throughout their life cycle. Once checked, the next function (`newLayer`) performs the information processing (i.e., load a new layer on the map). The function `receiveDeleteLayer` also follows the same process.

Once the set of components to be executed in our system has been constructed, they are integrated in a user interface. To do this, each user interface is constructed using a Web page where the components (to be executed at a given moment for a given user) are identified. Table 3 shows a fragment of the Web page `index.html` of the application described in this article. This page is loaded in the Samsung Smart TV when a user accesses the system. The web page shows all those components included under a `<body>` label, and by means of a `<div>` label. Label `<body>` represents the `Main Container` of a user interface. Label `<div>` includes as parameters: the `id` and the `class` of the component, being the latter the type of widget. This `<div>`-based structure provides a feature in the user interface because of the frame that is displayed around the visible components. It also enables the user to perceive the interface as a set of components that can be manipulated according to his preferences. Label `<iframe>`, which contains the concrete component identified by its `id` and linked by `src`, is also included under a `<div>` label. To delete a component from the interface, label `<div>` must be deleted.

Table 1: Structure of the Map Component in W3C Widget.

```
Map.wgt
  index.html  //Main File
  config.xml  //Config of resources
  content/
    interaction/
      UIscript.js  //Visual functionality
      images/
        header.png
    functional/
      Taskscript.js  //Data base functionality
```

Table 2: Example of Taskscript.js.

```
<script>
// connection to server
var websocket = io.connect('http://acg.ual.es:6969');

// connection to server with the component username
websocket.on('connect', function(){
  // call to server through function 'adduser' and
  // send necessary parameters
  websocket.emit('adduser', getUrlVars()['ID_user']);
});
websocket.on(
  'receiveLoadLayer',
  function(componentName, nameLayer, dataLayer){
if(componentName == 'Options'){
      newLayer(nameLayer, dataLayer); }}
);
websocket.on(
  'receiveDeleteLayer',
  function(componentName, nameLayer){
if(componentName == 'Options'){
     deleteLayer(nameLayer); }}
);
</script>
```

4.2 Server Side technology

As mentioned, we have developed a system comprised of several servers based on Web technologies. In Figure 5 may be observed that a server called Apache Wookie[7] has been deployed. This server is used to manage our component repository, making it possible to deploy, update and store components constructed following the W3C Widget specification.

In addition to a component managing server, we

[7]Apache Wookie: http://wookie.apache.org/

Table 3: Embedding Wookie & widgets into the main page.

```
<html>
 ...
 <body>
 <!-- Component Header -->
 <div id='Header' class='widget'>
   <iframe id='header'
    src='http://acg.ual.es/wookie/deploy/acg.ual.es/
    wookie/widgets/Title/index.html?idkey=
    5.pl.pu6bbHaBxkWRMWLyfd2m.sl.VMLw.eq.&
    proxy=http://acg.ual.es:80/wookie/proxy&st='>
   </iframe>
 </div>
 <!-- Component Elements -->
 <div id='elements' class='widget'>
   <div class='title'> <h4>Elements</h4> </div>
   <iframe id='ielements'
    src='http://acg.ual.es/wookie/deploy/acg.ual.es/
    wookie/widgets/Elements/index.html?idkey=
    .pl.z5.sl.ArbxzPbvESCH43BKZcw1fxI.eq.&proxy
    =http://acg.ual.es:80/wookie/proxy&st='>
   </iframe>
  </div>
  <!-- Component Political Map -->
  <div id='politicalMap' class='widget'>
    <div class='title'> <h4>Map</h4> </div>
    <iframe id='ipoliticalmap'
     src='http://acg.ual.es/wookie/deploy/acg.ual.es/
     wookie/widgets/MapOptions/index.html?idkey=
     gvXeT11dUC8JI40U9UetR.pl.gzhJ8.eq.&proxy=
     http://acg.ual.es:80/wookie/proxy&st='>
    </iframe>
  </div>
  ...
 </body>
</html>
```

used other server based on JavaScript called Node.js[8] to manage data in/out. This server is directed at events by making use of a non-blocking input/output model, which enables easy distributed construction of applications. The use of this server provides the capacity for making changes in the interface, such as loading, removing, resizing or redistributing a component within the interface itself (without reloading it).

To manage information sharing between components, we employed a communication by message exchange. In this communication method, the exchange of the message is done by means of "Node.js", which acts as an intermediary receiving the message of a component and sending it to the destination component. This communication behavior is part of the relationships mentioned above. Example of this kind of communication appears in the code shown in Table 2. As may be observed in line #3, the `Map` component is connected to the server providing to the `Map` with the information necessary to load or delete a given layer. To carry out this type of communication, our system uses a "Node.js" server *plug-in* called `socket.io`[9]. This *plug-in* assists in developing communications at run-time for applications located on different platforms.Finally, to insert a new component in a user interface this behavior is perfomed in the main page file of the user interface shown in Table 3. Assume we wish to insert the `Histogram` component and select it from the `Elements` component.

Figure 6 shows the sequence diagram describing the steps taken by the system to load the `Histogram` component. First, to initiate this process, the user should select the `Histogram` checkbox included in the `Elements` component (we can see this checkbox in Figure 3). Next, component `Elements` emits a message to `Node.js` server by means of the *socket.emit* function (message #2 in Figure 6). This function requires three parameters: the identifier of the port on which the message is issued (*addWidget* in this case), the owner of the graphical user interface (named as *user*), and the identifier of the component that must be inserted (*Histogram* in this case). Then, the `Node.js` server invokes the *addWidget* function of the Kernel system based upon previous information (#2.1). Sequentially, the `Kernel` calls the *createWidgetInstance* function of Wookie server (#2.1.1). Now, `Wookie` creates a widget instance of the *Histogram* (componentID) component for the *user* stated and returns the corresponding widget instance data for identification and usage purposes (#2.1.1 and #2.1.2).

Once this information is obtained, the `Kernel` builds the corresponding HTML code to be inserted into the user interface (#2.1.3). This code is returned to `Node.js` server (#2.1.4) and is emitted to the client application, that is, to the user interface (#2.2). Into the `MainContainer` component, there is a function in charge of listening the *addWidget* web socket port and that inserts the new code in the SmartTV application (#3). Finally, since the new HTML code references the widget instance residing in Wookie, the client performs a GET method call to the Wookie server in order to obtain the new widget instance (#4 and #4.1). The behavior of the running example can best tested in a SmartTV emulator available at http://acg.ual.es/enia/cbuismarttv.

5 RELATED WORK

In some studies such as (Teixeira-Faria and Rodeiro, 2011) the authors define a type of components called Abstract Interaction Object (AIO) which can describe an interface as a set of abstract components. In (Savidis, 2005) the author also describes a user interface as a set of AIO call Virtual Interaction Object (VIO), which gives an object definition that helps to developer to develop an object for different platforms. (Bodart and Vanderdonckt, 1996) describes other components, such as the Concrete Interaction Object (CIO),

[8]http://nodejs.org/

[9]http://socket.io/

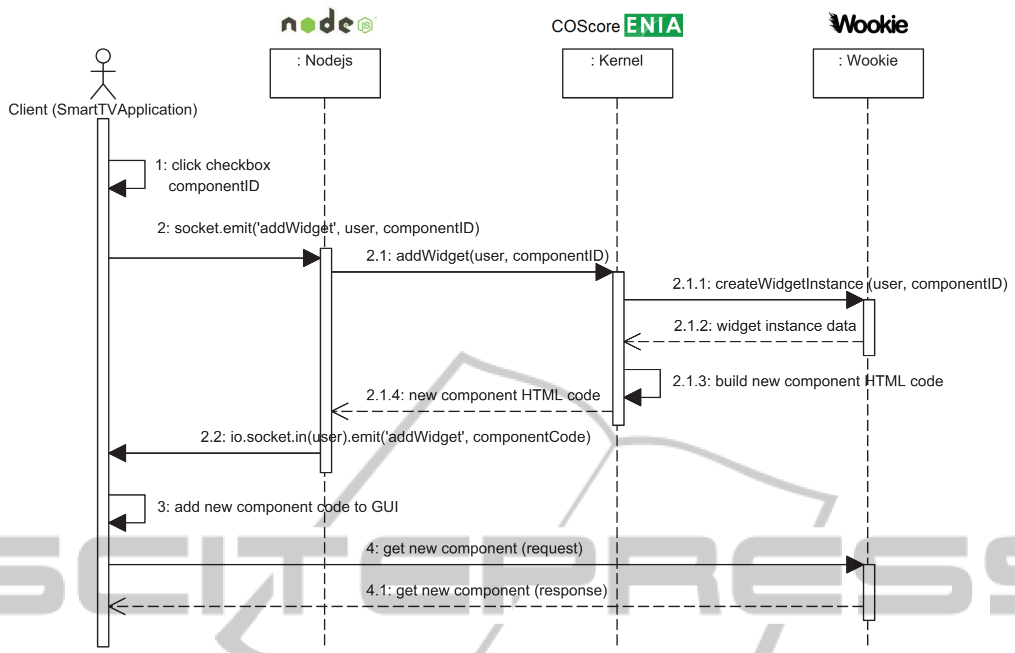

: Nodejs
COScore ENIA
: Kernel
Wookie
: Wookie
Client (SmartTVApplication)
1: click checkbox componentID
2: socket.emit('addWidget', user, componentID)
2.1: addWidget(user, componentID)
2.1.1: createWidgetInstance (user, componentID)
2.1.2: widget instance data
2.1.3: build new component HTML code
2.1.4: new component HTML code
2.2: io.socket.in(user).emit('addWidget', componentCode)
3: add new component code to GUI
4: get new component (request)
4.1: get new component (response)

Figure 6: Operation sequence.

which are defined as visual user interface components, and are used to display and manipulate information in database systems. However, none of these works use a CBA perspective for SmartTV.

In previous studies in (Iribarne et al., 2010) and (Iribarne et al., 2011), a CBA of widget-type components is proposed. These components are applied to a Web-based Information System, WIS, because of their flexibility, adaptability, accessibility and manageability by different people or groups of people with interests in common and different profiles located in different places. This kind of interface provides the possibility of reorganizing the interface components according to the user needs, improving the user experience in performing his tasks. Furthermore, in previous work, it was initiated the development of component-based systems to be deployed on the TV platform (Fernandez and Iribarne, 2010; Vallecillos et al., 2012). The present paper is based on both studies but improves the component and relationship definitions, and also provides a real implementation of Smart TV interactive system.

Finally, the development of software technologies for smart TVs is an evolving field, where component-based interface proposals have not yet been developed. Some of them propose the development of specific applications for television and others using standard television broadcasts add advanced services to the broadcast. One example of the first trend is the *Multimedia Home Platform*, MHP (Martin et al., 2010) for application development, which defines a common platform (middleware) for interactive smart TV applications, regardless of interactive service provider or type of television where it is executed. Some examples of the use of this platform are (Blanco-Fernández et al., 2008) and (Pazos-Arias et al., 2006). Another platform based on the second proposal is *Hybrid Broadcast Broadband TV*, HbbTV (Kuzmanovic et al., 2012), which uses standard wideband television broadcasts to show user entertainment services, such as games, social networks or interactive advertising using Web technology.

6 CONCLUSIONS AND FUTURE WORK

Nowadays there is fast growth of devices on different technological supports that enable the public access to IT services/applications in any personal environment. However, to construct these products, techniques for their development must be available.

Component-based Software Engineering (CBSE) is a software engineering discipline that facilitates this labor by focusing on the integration of previously constructed software components for systems. Our research concentrates on the development of software based on a CBSE perspective for a number of environments, such as the application described in this article for the Samsung Smart TV environment.

In addition, we have described a sort of component based GUI developed for an interactive Smart TV system. Furthermore, this system serves as a running example used through the paper. We presented some component and relationship issues and we exemplified them by using the previous GUI. We explained the technological solution adopted to implement our architecture in the Smart TV environment. In order to better understand the interactive Smart TV system, we created a web page (http://acg.ual.es/enia/cbuismarttv/) where some other information about this work is available, including the emulator installation process, the Samsung Smart TV project developed, and a video where the interaction with the application is showed.

As future work, we will carry on a study of the users and user profiles registered in the system, which could be useful to trace user interaction. With this information we will be able to create interactive systems adapted to the user's needs. In order to improve the justification of the proposed method, we intend to conduct a controlled experiment with different groups and different kinds of users. Each user or group could perform some development part and then fill in a survey form, for example, by comparing our approach with traditional methods to develop GUIs or component-based GUIs. On the other hand, we want to extend our system with some new functionalities to provide a support for cooperative tasks. Finally, we intend to deploy our system in other platforms such as tactile devices, or use other interaction methods such as Natural User Interfaces (NUI).

ACKNOWLEDGEMENTS

This work was funded by the Spanish Ministry of Economy and Competitiveness (MINECO) under Project TIN2013-41576-R and the Andalusian Regional Government under Project P10-TIC-6114 and the Spanish Ministry of Education, Culture and Sport (MECD) under a FPU grant (AP2010-3259). This work was also funded by the CEiA3 and CEIMAR consortiums, and the Applied Computing Group.

REFERENCES

Belli, F. (2013). Dependability and software reuse – Coupling them by an industrial standard. In: IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C), pp. 145–154.

Blanco-Fernández, et al. (2008). An MHP framework to provide intelligent personalized recommendations about digital tv contents. *Software: Practice and Experience*, 38(9):925–960.

Bodart, F. and Vanderdonckt, J. (1996). Widget standardisation through abstract interaction objects. *Intitut d'Informatique, Facultes Universitaires Notre-Dame de la Paix, Namur, Belgium*.

Criado, J., Iribarne, L., and Padilla, N. (2013). Resolving Platform Specific Models at runtime using an MDE-based Trading approach. IY.T. Demey and H. Panetto (Eds.): OTM 2013 Workshops, LNCS 8186, Springer, pp. 274–283.

Crnkovic, I. and Larsson, M. (2001). Challenges of component-based development. *Journal of Systems and Software*, 61 (3):201–212.

Fernandez, A.J., and Iribarne, L. (2010). TDTrader: A methodology for the interoperability of DT-Web Services based on MHPCOTS software components, repositories and trading models. Proc. 2nd Int. Workshop of Ambient Assisted Living, (IWAAL2010), pp. 83–88.

Iribarne, L., Criado, J., Padilla, N., and Asensio, J. (2011). Using COTS-widgets architectures for describing user interfaces of web-based information systems. *Int. Journal of Knowledge Society Research*, 2(3):61–72.

Iribarne, L., Padilla, N., Criado, J., Asensio, J., and Ayala, R. (2010). A model transformation approach for automatic composition of COTS user interfaces in web-based information systems. *Information Systems Management*, 27:207–216.

Kuzmanovic, N., Mihic, V., Maruna, T., Vidakovic, M., and Teslic, N. (2012). Hybrid broadcast broadband tv implementation in Java based applications on digital TV devices. *IEEE Transactions on Consumer Electronics*, 58(3):1056–1062.

Martin, C. A., Garcia, L., Menendez, J., and Cisneros, G. (2010). Access services based on MHP interactive applications. *IEEE Transactions on Consumer Electronics*, 56(1):198–202.

Pazos-Arias, J. J., López-Nores, M., García-Duque, J., Gil-Solla, A., Ramos-Cabrer, M., Blanco-Fernández, Y., Díaz-Redondo, R. P., and Fernández-Vilas, A. (2006). Atlas: A framework to provide multiuser and distributed T-learning services over MHP. *Software: Practice and Experience*, 36(8):845–869.

Remagnino, P., Hagras, H., Monekosso, N., and Velastin, S. (2005). Ambient Intelligence Springer New York, pp. 1–14.

Savidis, A. (2005). Supporting virtual interaction objects with polymorphic platform bindings in a user interface programming language. *LNCS*, 3475:11–22, Springer-Verlag Berlin, Heidelberg.

Teixeira-Faria, P., and Rodeiro, J. (2011). Complex components abstraction in grapphical user interfaces. *Human-Computer Interaction, Springer*, pp. 309–318.

Vallecillos, J., Fernndez, A.J., Criado, J., and Iribarne L. (2012). TvCSL: An XML-based language for the specification of TV-component applications. Communications in Computer and Information Science, Springer Vol. 278, pp. 574–580.