Framework for Graphical User Interface Personalization in Interactive Television

Tomislav Ćurin, Hrvoje Bašić, Mario Žagar
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
tomislav.curin@fer.hr ; hrvoje.basic@fer.hr; mario.zagar@fer.hr

Abstract. *Along with the inevitable digitalization of television, new interactive services are becoming available. One of the vital parts of such interactive applications are their graphical user interfaces. This paper describes the framework for more efficient development of personalized graphical user interface for interactive television applications based on the MHP standard.*

Keywords. Interactive television, MHP, personalization, framework, graphical user interface.

1. Introduction

The breakthrough of digital television encourages the introduction of interactive services and applications which can be broadcasted along with the television program. Numerous possible applications come to mind when interactive television (iTV) is mentioned. Electronic program guides, various information services (i.e. super teletext, stock tickers), video-on-demand, e-commerce and home banking, gaming and applications synchronized to TV content are some of the most typical examples. The spectrum of possible services is not limited to those, in the future it will be extended to many others, some of them not yet conceivable today.

As every new opportunity brings new challenges, it is also the case with interactive applications. Among many concerns that arise is the construction of the graphical user interface. The user interface, as the part of the system that is directly exposed to the user, is one of the key elements in the application design, and its role is even more important in iTV applications. An application that seems to be too simple may be considered boring, uninteresting, and too business-like. This is why special attention has to be focused on the pleasant appearance of the application [9].

However, designing the user interface for television is something that is quite different from PC graphical user interface design. As noted in [7], user interface development is more complex for this type of platform when compared to that for a desktop computer as many constraints are set on the look and feel. The use of complex graphical interfaces is limited because television screens have low resolution compared to PC monitors. Furthermore, some colour combinations that look good on a PC monitor don't display well on a TV screen, and certain backgrounds tend to display distorted and unreadable [6]. The lack of input devices like a mouse, and most of the times even a keyboard, which are common tools in PC systems, just adds to the complexity of the user interface design for iTV applications. Developers can only rely on the fact that the typical end user is equipped with the remote control.

Another issue that comes along with iTV is indispensable internationalization of the applications. From the developer's point of view supporting one language (e.g. English) would be preferred as it requires less work and is cheaper to produce. From the users' point of view, however, the possibility to use menus in their own language is a huge benefit [3].

Despite the tremendous potential, the uptake of iTV by consumers is still relatively slow. To ensure broad adoption, digital television will have to take usability issues thoroughly into account [1]. Personalization through adaptive graphical interface and multilingual support are some of the factors that would aid better usability. Since the support for development of the iTV applications is the area which has yet to grow, implementation of these elements is usually not an easy task at all. Therefore, our aim was to create a framework that would support simple realization of the mentioned elements. This paper deals with the description of the created framework, specifically designed for the

Multimedia Home Platform – MHP – which is emerging as the internationally accepted standard for interactive television.

2. Multimedia Home Platform

Over the past couple of years, the Digital Video Broadcasting consortium (DVB), as the leading standardization group in digital television, has put lots of effort into creating an open standard platform specification for interactive digital TV and multimedia services. Its aim was to standardise elements of the home platform (set-top box, television etc.) that would be the key to the success of interactive multimedia applications in the future [2]. Such Multimedia Home Platform – MHP – would allow portability of interactive television content and enable broadcasters and operators to provide universal services for all compatible decoders, ranging from low-end to high-end set top boxes and integrated digital TV sets.

The technical specification of the MHP has been developed against a comprehensive set of commercial requirements that address the needs of both commercial and public sector broadcasters [5]. It was intended that MHP would facilitate the development of an open, horizontal market for vendors, authorities and broadcasters. In the year 2000, DVB eventually succeeded to complete the MHP specification.

MHP defines a generic interface between interactive digital applications and the terminals on which those applications execute. It consists of a user terminal (PC, integrated TV, digital set-top box, associated peripherals, in-home digital network), a standard middleware, and a suite of APIs that are capable of supporting a range of services [2].

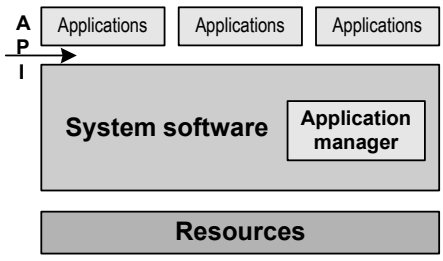

Figure 1: MHP Layers

The architecture of an MHP set top box can be considered in terms of three layers (Fig 1):

- Resources
- System software
- Applications

The *resources* layer is responsible for MPEG processing, I/O devices, CPU, memory and graphics system. *System software* layer accesses available resources to provide an abstract view of the platform to the *applications*. Implementations include an application manager to control the MHP and the applications running on it [11].

DVB has also decided to adopt Java technology and to base MHP around DVB-J, a subset of Java which includes a virtual machine as defined in the Java Virtual Machine specification and also provides extensions which are suitable for interactive TV applications. Standard Java authoring tools may be used to design DVB-J set-top box applications.

2.1. Graphics on MHP

MHP graphics architecture is composed of three different overlapping planes, from back to front: background, video and graphics (Fig 2).

An application is provided with a contiguous rectangular region of the graphics plane in which it can draw, place video, interface elements and graphics. Applications can also control video on the video plane outside of the graphics plane, and place still images, video drips or solid colour in the background plane [2].

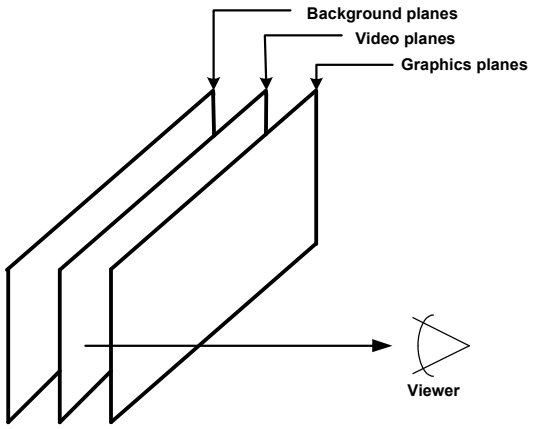

Figure 2: Illustration of the different types of display planes [2]

Graphics API on the MHP comes with the significant design constraints as Swing classes, which are usually used for GUI design in Java applications, are not included in the MHP standard. The graphical user interface API defined in MHP is based around a subset of Java Abstract Window Toolkit (AWT) which includes only graphics primitives and event handling. It does not include heavyweight widgets such as buttons and text fields. To aid user interface

design, MHP API also encompasses Home Audio/Video Interoperability Level 2 User Interface extension (HAVi), which allows Java applications to determine the user interface capabilities of its host display device, accept input from the user, draw to the screen and play audio clips [12]. HAVi includes ready-made components specially designed for usage on TV screens and also solves issues related to video and graphics integration.

Although HAVi user interface implementation should be the same on all set-top boxes, the layout of the components is sometimes very different with the different receivers [9], and in our experience, on some receivers not valid at all. This is a substantial issue, especially as the broadcasters consider it very important that they are able to precisely control the look and feel of their applications and services, and that if they cannot control the look and feel of the resident widget set, they will not use it [8]. In addition to compatibility obstacles, an application using HAVi components typically functions slower than an application with the user interface components implemented using the Java language [9].

Given the state of facts, we decided to build our own GUI library of custom made widgets using AWT primitives. However, HAVi was not completely bypassed, and some fundamental parts of it were utilized. We also decided to build a framework along with the library which would allow easy configuration of application look and feel and support for multiple languages. In addition, an essential requirement was that the framework provides a possibility to alter the graphical user interface during runtime, without the need to recompile the application.

3. Framework description

The framework consists of two independent parts, *component interface* – a GUI library which contains a widget set for general usage, and the *skin interface*, which provides mechanism for implementation and distribution of custom look and feel as well as multilanguage support.

3.1. Component interface

The idea to circumvent usage of HAVi and to create application specific widgets is not a novelty, quite opposite, it is probably the most popular method for the applications to provide their own look and feel [4]. However, this approach has some drawbacks, most important, such widgets typically cannot be reused in other applications.

In our GUI library we have created a standard widget set which can be extended at a later time if needed. Applications can use the widget set to draw graphical components on the screen as well as to receive and to respond to user events from interactive devices (e.g. remote control or keyboard). Library implementation is based on AWT and follows HAVi specification. Base classes, *AComponent* and *AContainer* are derived from HAVi *HComponent* and *HContainer* classes. Component interface hierarchy is presented in a diagram on Fig 3.

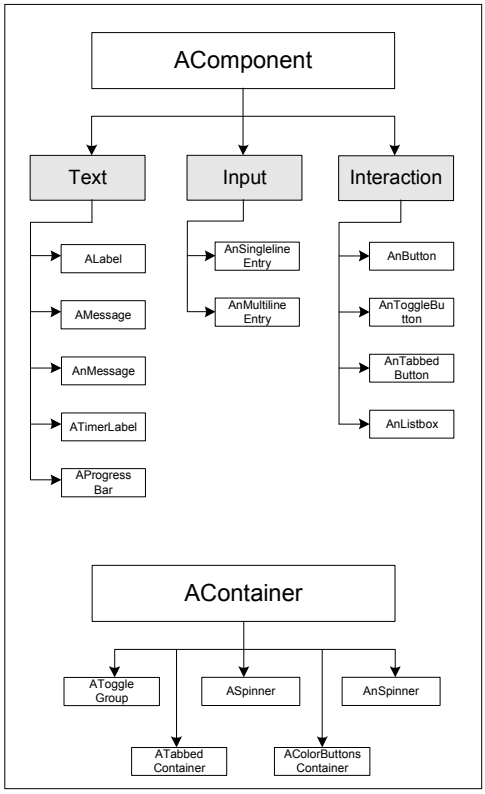

Figure 3. Component hierarchy

Following groups of widgets have been developed:

- Text widgets
- Input widgets
- Action widgets

Text widgets can display text in one or multiple lines. User interaction is needed only to scroll text down or up if text dimensions exceed the widget size. Input widgets provide means for users to input text in one or multiple lines. Action widgets are various types of buttons (e.g. simple text button or toggle button), which allow users to induce different types of actions.

In order to be consistent with the HAVi specification for interaction states, components are divided into two groups:

- Non-navigable components, which cannot acquire focus and can only enter two states: enabled or disabled.
- Navigable components can acquire focus, and can have extended number of states.

The component's visual presentation on the screen is tightly related to its state. Every framework component must have an associated class which implements rendering methods for fill, border and text effect. Following default classes are implemented: *FillEffect*, *BorderEffect* and *TextEffect*. *FillEffect* determines background fill method, *BorderEffect* renders border if one is needed, and *TextEffect* is used to set text preferences. When necessary, *paint* method in one of these classes can be overridden in order to implement a specific component look. Once defined, effect class can be reused on every other component with the same number of states.

In order to respond to events, components keep their internal list of registered event listeners. When events occur, a component notifies all listeners about it. The event model follows that of HAVi's, with some extensions. Since actions in MHP environment (e.g. local data loading, downloading from a broadcast stream, navigating through the application) are often quite slow, we found it convenient to provide a mechanism for indication to the user that an event is being processed. For that purpose, apart from the standard HAVi events, we introduced a new event, called *ProgressEvent*, which conveys progress of an action. Components registered as progress listeners are notified about action progress: its start, change and stop. The component can, depending on its type, display progress status or perform any other action as needed.

Lastly, several helper classes have also been implemented. Mainly container classes, they implement appropriate grouping of toggle buttons (e.g. toggle group or tabbed container).

3.2. Skin interface

One of the most valuable capabilities of the Swing toolkit is its pluggable look and feel architecture, which allows seamless changes in the application appearance and the way an application interacts with the user (i.e skin). This can occur without modifying or recompiling the application and it can be invoked programmatically during any single JVM session [10]. However, as already noted, Swing is not incorporated in the MHP specification, so unfortunately this feature cannot be utilized.

HAVi user interface contained in the MHP standard includes *HLook* interface, which defines the mechanism by which the content for the particular state of the widget can be rendered [12]. This should allow easy construction of many presentation styles. However, it is very inconvenient to define new look class for different component types. Also, HAVi lacks support for non-programming modifications of user interface. The possibility to define skin through a property file was one of the main objectives of this framework. Besides that, strong arguments for avoiding HAVi were already mentioned compatibility problems.

In our framework implementation, the skin interface is used to load visual properties of the application. Attributes of the widgets contained in the application can be defined in the property file and loaded using Java Reflection API. It is possible to define every widget property which can be accessed through the setter method.

Main features of the skin interface are:

- Loading of images and sounds
- Splitting larger images into smaller chunks (loading sequence of images)
- Multiple language support
- Possibility to define data arrays

Components with visual identity defined in the property file which contains skin data are created in several stages:

- *Properties loading*. Properties from the configuration file are loaded into the properties table.
- *Property objects creation*. Property objects are created by iterating through the values contained in the properties table from the first step.
- *Widgets creation*. Visual component objects are instantiated, and their visual characteristics are specified using property objects from the previous stage.
- *Widgets usage*. Finally, component can be used in the code and added to the application's graphics plane.

An application can also load a still background image specified in the configuration file and place it in the background plane as defined in the MHP specification [1].

Lastly, multilanguage interface was a considerably simpler task to accomplish as MHP includes *ResourceBundle* class of the Java API which provides means to detach translatable text and localizable objects from the source code. Therefore only a wrapper class had to be realized for retrieving and formatting of localized data. One problem that remained is that text size of the expressions can vary among the different languages. Therefore, all the widgets have additional boolean property called *resizeToText*. Setting this flag to true will instruct widget to automatically change its size and position on the screen depending on the calculated text length.

4. Prototype application

A prototype of the message client application was built using the described framework. Properties of all widgets used in the application were defined and loaded using the framework.

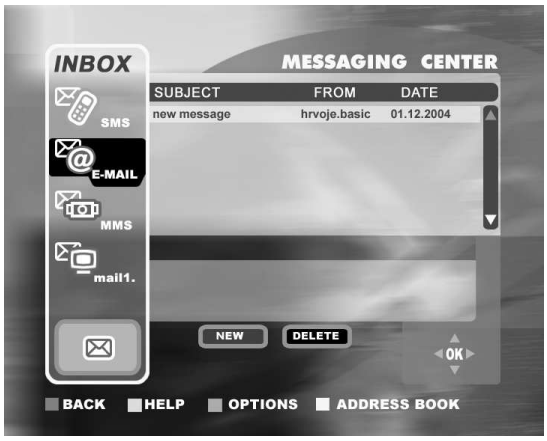

Figure 4. Prototype application

The layout of the application's main screen was divided into four logical sections (Fig 4). A section that contains image buttons (SMS, MMS or E-mail) grouped into single radio button group was placed in the left part of the screen. By selecting one of these buttons, the user can select desired message type. A list box containing messages kept in the current inbox was placed at the screen centre. After selecting message type, the user can scroll through the inbox list, select and read a message. Action buttons for deleting the message or creating a new one are positioned in the lower central section. Finally, at the bottom of the screen, there is a static container with coloured and text labels representing red, green, blue and yellow buttons of the remote control. Those buttons are used for navigation through other application screens.

By specifying a new property file and defining a few new images, we achieved an entirely different application look and feel without any code modifications. As it can be seen on Fig 5, sections and all widgets had their screen position altered. Beside position changes all widgets have a different appearance. In addition, the background image was modified, and the application language was set from English to Croatian.

Figure 5. Prototype application - modified

The possibility to make such look and feel changes during application runtime opens up a path for a scenario where the user is allowed to select among several predefined skins and immediately see the results of his selection. User selection can then be stored in the database on the server side of the application as the user's personalized choice and loaded as the preferred skin each time she or he logs in. Extending the application with a new user interface should then be a simple task of two steps:

- define a new skin in a configuration file and prepare it for broadcast
- update the database on the server side with information about the skin

5. Conclusion

The need for the described framework emerged as a consequence of the important demand to provide users with the way to customize application to their own needs. Methods of application customization include prespecified adaptive graphical user interface and multilingual support, which is beneficial in an interactive television environment. Fulfilling these requirements would encourage wider adoption of interactive television services among potential users. We believe that these elements

could be efficiently realized with the proposed framework. There are, however, many other advantages of the framework from the developer's perspective as well.

Visual identity of an application that utilizes the framework is separated from the other parts of the application (as it is defined in the properties file), which in turn makes it possible to change application appearance during runtime. That is quite valuable to the user, who can select the preferred look and feel and promptly see the results of his or her choice. It is of even more value to the developer, who does not need to make code modifications each time he or she wants to adjust application GUI.

Moreover, reusability was one of our main objectives, so none of the widgets contained in the library has application specific features and can be used in any other application. Reusability approach in the way the components are being defined in the skin interface has significantly reduced application's memory footprint. All of the mentioned benefits are of great importance during the development and testing phase in a set-top box environment, which is typically limited in memory and processor speed. Finally, it is worth noticing that the widget library, implemented as part of the framework, can be easily extended with other widgets, either general purpose or application specific.

The planned framework extension is usage of XML file instead of Java property file. This would provide easier skin definition and better portability, but the drawback of this solution would be the larger burden on the set-top box resources as parsing of an XML file is a process that takes more memory and processor time. Possible workaround to this problem would be to create a tool which would pre-compile XML file into a format specially prepared for the application (e.g. binary file). Other possible extensions are related to the widget library which could be extended with some commonly used graphical components, namely horizontal and vertical tickers, and virtual keyboard. Finally, an IDE (e.g. Eclipse) plugin for visual editing of GUI and automatic creation of a skin property file would be an extension potentially of a great benefit, although its development would probably be a challenging task.

6. References

[1] Ardissono L, Kobsa A, Maybury MT, editors. Personalized Digital Television: Targeting Programs to Individual Viewers (Human-Computer Interaction Series Vol. 6). New York: Springer; 2004

[2] Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.0.3. DVB; 2000. http://www.mhp.org/index.php?id=216 [02/28/2005]

[3] Fernström J. Commercial Graphical User Interface for Digital TV, Master Thesis in Human Computer Interaction. Stockholm: Centre for User Oriented IT Design, 2004; http://cid.nada.kth.se/pdf/252.pdf [02/28/2005]

[4] Morris S. Graphics APIs. http://www.mhp-interactive.org/tutorial/mhp/graphics.shtml [02/28/2005]

[5] Newell JC. The DVB MHP Internet Access Profile. BBC; 2002. http://www.bbc.co.uk/rd/pubs/whp/whp-pdf-files/WHP018.pdf [02/28/2005]

[6] O'Driscoll, G. The Essential Guide to Digital Set-top Boxes and Interactive TV. Upper Saddle River, NJ: Prentice Hall; 2000.

[7] Peng C. Digital Television Applications, Doctoral Dissertation, Helsinki University of Technology, Espoo, Finland, 2002. http://lib.hut.fi/Diss/2002/isbn9512261723/ [02/28/2005]

[8] Peng C, Vourimaa P: Development of Java User Interface For Digital Television Proceedings of the 8th WSCG International Conference on Computer Graphics, Visualization and Interactive Digital Media, 2000 Feb 5-9, Plzen, Czech Republic. p. 120-125.

[9] Rinnetmäki M, Heikkinen M, Kosonen I, Saarijärvi M, Pöyhtäri A, Nykänen P, Saikanmäki A. A Guide for Digital TV Service Producers. ArvID; 2004. http://www.arvid.tv/java/Index?oid=84 [02/28/2005]

[10] Robinson M, Vorobiev P. Swing 2nd edition. Greenwich: Manning Publications Company; 2003.

[11] Smith-Chaigneau A. DVB-MHP - A Snapshot. DVB Project Office, 2001. http://www.broadcastpapers.com [02/28/2005]

[12] The HAVi Specification, Specification of the Home Audio/Video Interoperability (HAVi) Architecture, Chapter 8 – Level 2 User Interface; version 1.1; 2001. http://www.havi.org [02/28/2005]