AMBIENT ASSISTED LIVING, AAL

JOINT PROGRAMME

ICT-BASED SOLUTIONS FOR ADVANCEMENT OF OLDER PERSONS' INDEPENDENCE AND PARTICIPATION IN THE "SELF-SERVE SOCIETY"

D3.2

TV Client

Project acronym:	**GeTVivid**
Project full title:	**GeTVivid - Let's do things together**
Contract no.:	**AAL-2012-5-200**
Author:	**IRT**
Dissemination:	**Public**

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY 4

1.1 LINK WITH THE OBJECTIVES OF THE PROJECT 4

1.2 STATE OF THE ART 4

2. HBBTV PLATFORM OVERVIEW 5

3. TECHNICAL INFRASTRUCTURE 8

3.1 DISTRIBUTION SCENARIO 8

3.2 TV CLIENT PROFILES 8

4. PROTOTYPING 11

4.1 SCENARIO IMPLEMENTATION 11

4.2 CLIENT IMPLEMENTATION 12

4.2.1 Page Layout 13

4.2.2 Navigation 15

4.2.3 Input 18

4.2.4 Dialogs 21

4.2.5 Lists 25

4.2.6 Second screen 27

4.2.7 Video player 30

4.2.8 Notifications 34

5. OVERALL CONCLUSION 36

6. LITERATURVERZEICHNIS 37

7. ANNEX A 38

8. ANNEX B 42

TERMINOLOGY & ABBREVIATIONS

CEA ..... Consumer Electronics Association

CSS..... Cascading Stylesheets

DVB..... Digital Video Broadcasting

ETSI..... European Telecommunications Standards Institute

HbbTV..... Hybrid Broadcast Broadband TV

HTML..... Hyper Text Markup Language

Wi-Fi ..... Wireless Fidelity

OIPF ..... Open ITV Forum

PVR ..... Personal Video Recording

STB ..... Set-Top-Box

UI ..... User Interface

iTV ..... Interactive Television

1. EXECUTIVE SUMMARY

1.1 Link with the objectives of the project

This document describes the current revision status of the iterative development and design process of the TV client based on the user and business requirements derived from T2.2 and T5.2. After giving a short overview of the HbbTV standard, which was used as a platform for the TV client, it addresses the imperatives of the client in respect to the elderly users and their living conditions. The main aspect of this document is to show the development process of the HbbTV client.

Foremost three TV client profiles are presented (broadcast, interactive and interactive with a second-screen) which take care of the different preconditions referring to the differing network linkage and usage of input devices in the user's home. The paragraph is followed by an outline of how the client was implemented, mainly consisting of the frontend development and the communication with the server. When designing the frontend, various HbbTV specific issues have to be considered, which were addressed by adhering significant iTV guidelines. For example the navigation and user input with a remote control pose a new challenge to elderly users. In conclusion the up-to-date efforts which were performed in WP3 relating to the TV client are summarized.

1.2 State of the art

Over the last years the communication among people through the web has changed our daily lives vastly. The distribution of internet capable devices and its network linking has reached almost all countries in the world. While the possession of PCs, tablets and smartphones has become common for the younger and middle-aged generation, older people still often seem to avoid new promising technologies, because they fear that they're not up to the complexity of its usage. As the internet now engrosses more and more devices, which were already present before the advancement of the World Wide Web, this seems to become a possibility for the older generation to take part in the internet community.

Due to the circumstances of the usage environment, e.g. on a couch via a remote-control, a system brought to the senior citizens home without investing in expensive new hardware might be a crucial factor for the acceptance of those new technologies. As the scope of GeTVivid is to assist elderly people with mildly impairments, so that they can benefit from upcoming technologies in order to facilitate their lives, this could be a step in the direction of integrating our older generation. As different technology standards have emerged and also disappeared over the last decade, the European market is now finally adopting the HbbTV standard as a common basis for application development on television sets. As the market penetration of HbbTV is constantly growing and almost all elderly people own a TV, the GeTVivid consortium decided to use this technology enabler as a basis for the development of the TV client. The following section shall give an overview about the Hybrid TV standard HbbTV as a technology enabler, before going into detail with the design and development of the GeTVivid TV client application.

2. HBBTV PLATFORM OVERVIEW

One of the first interactive TV systems in Europe, which was available in the 80ies, was called BTX (Bildschirmtext, screen-text). This system required a television-set for the presentation on screen and a telephone with a modem to transfer the data. Later at the turn of the century further proprietary systems like Mediahighway or Open TV evolved, fostering the development and need for an API standard. MHP (Multimedia Home Platform) was the first to achieve that, but had very limited success in the market. With the advance of the internet accompanied by broadband, becoming present to everywhere and everybody, and its capabilities regarding the use of multimedia-content, new players like Apple and Google are also trying to get their share of the TV-market in Europe by releasing set-top-boxes in order to distribute their content via the television.

When the first generation of Smart TVs entered the world-market, the majority of TV-manufacturers had its proprietary solutions which led to technological fragmentation and access issues (services had to be adopted for each device individually). Secondary there was no real “hybrid” approach to “connected TVs”. Therefore, the core group of the HbbTV initiative formed in 2009 including major manufacturers like Samsung and Sony, broadcasters, telcos, service provider, and research institutes like the IRT.

It had its roots in Europe and was designed in the context of DVB, focusing on the European broadcast market and the related regulatory requirements. The overall design targets were a pragmatic compromise between functionality, hardware and implementation effort by using existing standards as much as possible. The norm combined different standards from the broadcast and broadband world (DVB, OIPF, CEA, W3C, etc.) and was extended by further norms in the later versions of HbbTV (CI+, DRM). The outcomes were fixed in the HbbTV specification (see Figure 1), which was later published as an ETSI norm (ETSI, 2012). Interoperability with other interactive platforms was not covered in the original specification release.

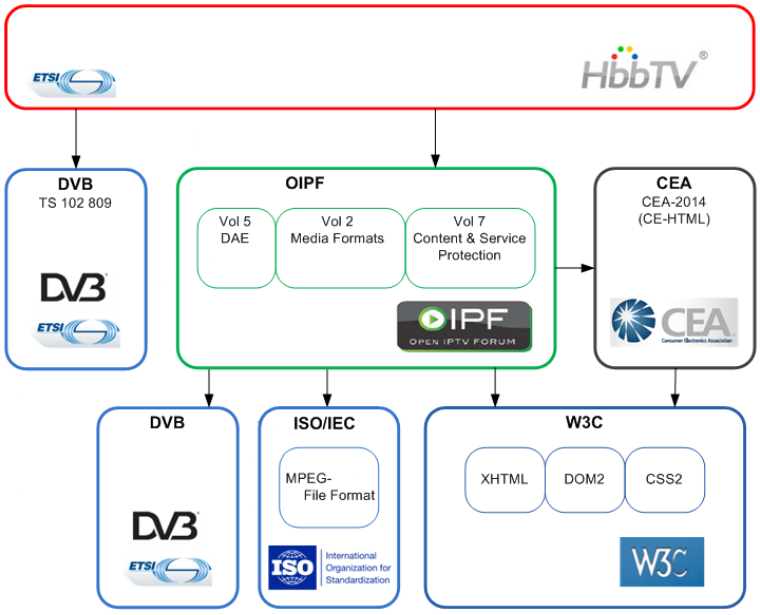

Figure 1: HbbTV 1.x specification overview (ETSI, 2012)

One of the major goals was to create an open standard, which was not based on a single controlling authority. In addition services from different and independent providers shall be accessible as well as the integration of broadcast-related and –independent applications. The playout of HbbTV services is fully network agnostic. The user can access the internet-data either via DSL or cable and the broadcast network can be delivered via satellite, cable, terrestrial or DSL.

Further issues were the support of various hardware platforms (e.g., IDTVs, set-top-boxes, and PVRs), the seamless combination of broadcast and broadband content, and the replacement of conventional Teletext services. The development of HbbTV-applications is based on HTML, CSS and Java Script; hence web developers were able to use "their" language to enter the broadcast world. The main adaption, compared to a regular website, was the creation of specific browser-profiles for the TV usage.

The most important advancement of the HbbTV standard (2.0) is the integration of selections from HTML 5 and other next generation web technologies, as well as the support of second-screen devices allowing a platform-independent user experience. Further enhancements were made in the fields of adaptive streaming and advanced graphics. Due to the customers replacing their old TV-sets over time, more and more devices are entering the market with HbbTV (see Figure 2).

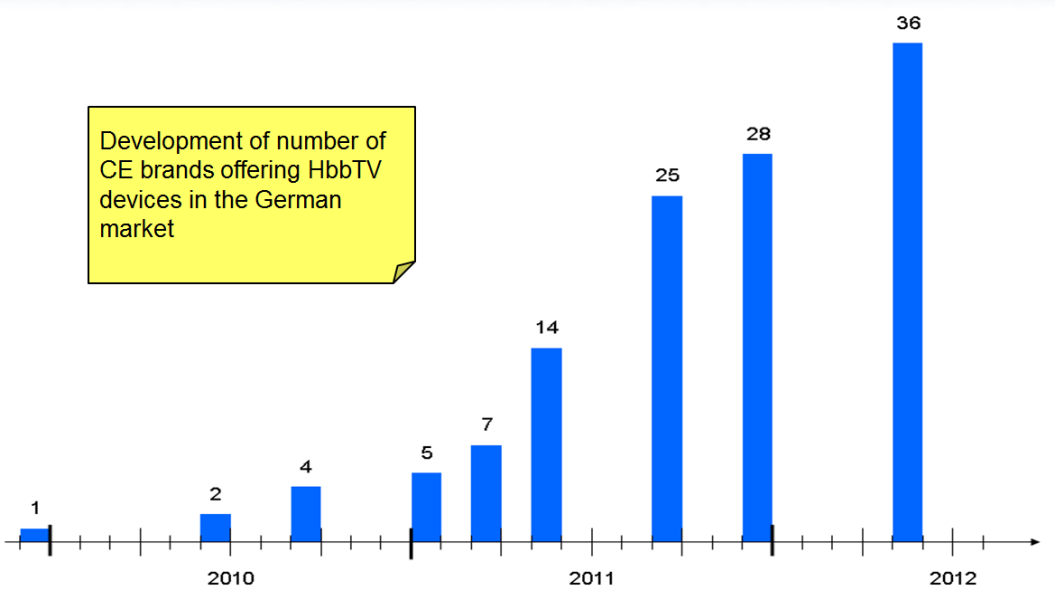

Figure 2: HbbTV expansion rate (K. Merkel, 2014)

In Europe HbbTV became the leading standard for Hybrid-TV covering ten countries and more will presumably be joining the HbbTV-community in the next years (see Figure 3).

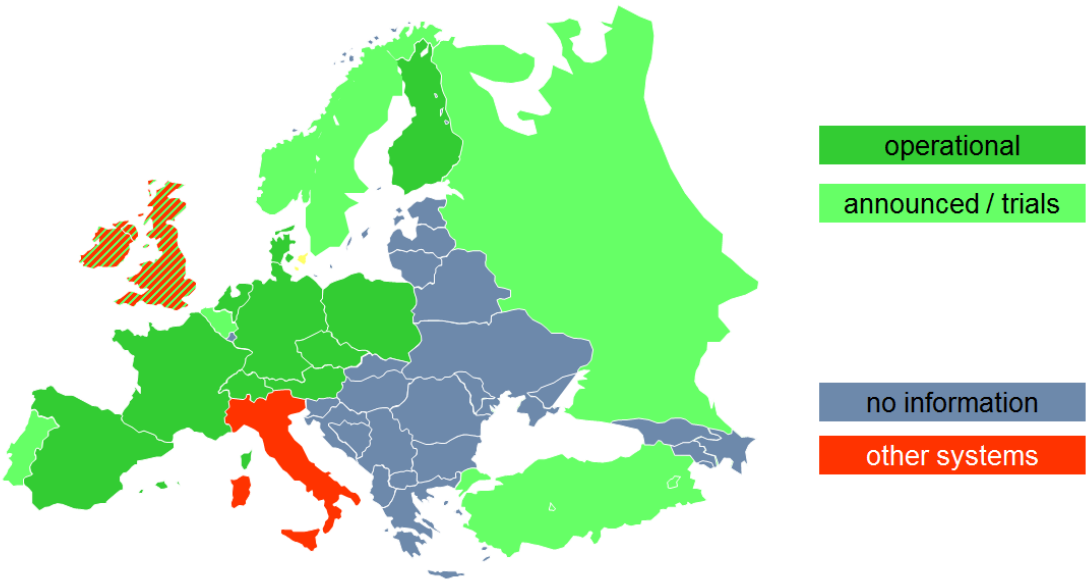

Figure 3: Geographical extension in Europe (K. Merkel, 2014)

3. TECHNICAL INFRASTRUCTURE

After presenting a short overview about HbbTV, the next section shall inform about how the TV client can be integrated in the home of older adults. Different options are presented, taking care of the technical, but also living conditions in the user's home.

3.1 Distribution scenario

The GeTVivid application shall become a horizontal platform tailored for older adults, targeting institutions such as retirement homes, but also individuals living at home. As part of the business plan, it has to be thought of how to integrate the application in the infrastructure of those premises. There are technically two possibilities to do so:

- The application is part of an application portal integrated in the TV set or set-top-box hosted by a device manufacturer.
- The application is played out via the TV signal of a local broadcaster. It can also be part of institutional premises just receivable for the older adults, i.e., local TV distribution system in retirement homes.

Even when assuming that all users in such a closed system have an HbbTV capable TV set, several scenarios have to be taken care of.

3.2 TV client profiles

An HbbTV enabled device receives data via two different ways: Firstly via the conventional broadcast DVB-signal (cable, satellite or terrestrial) and secondly via an internet connection either via tethered or Wi-Fi access. The web connection also allocates a backchannel for the client, making it possible to provide interactive services to the user. Even older devices without internet link can be upgraded with a so-called set-top-box connected to the TV, making it easy to adapt to the requirements of internet-enabled television without replacing the whole TV set.

But as not all households targeted will have an internet connection, it was decided to create three different access profiles to adjust to the needs of the users according to their access preconditions in their homes:

- **Broadcast** (no user input): only broadcast access without internet access; this is a one-way communication and restricts the range of available services, i.e., the user cannot send messages or book services, but they can receive messages via a dedicated data stream in the broadcast signal.

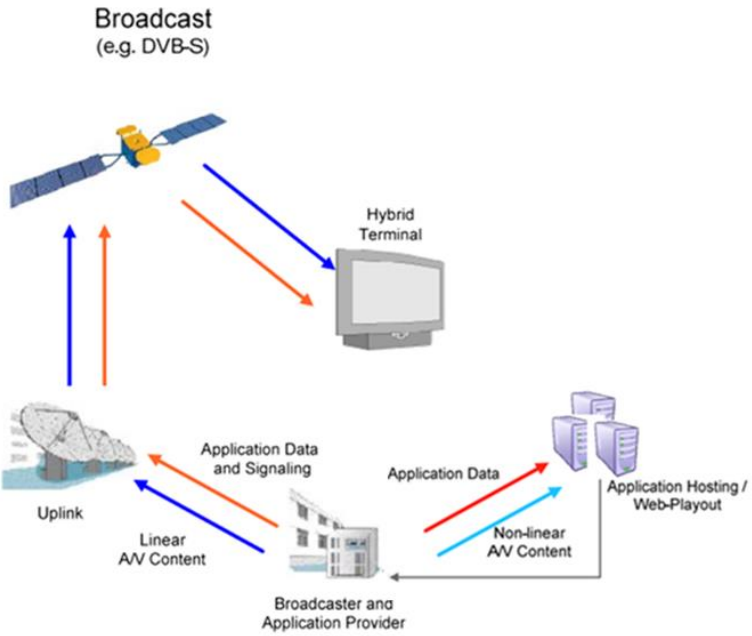

Figure 4: Broadcast profile

- **Interactive** (user input via remote control): broadcast and internet access; using the backchannel for interactive services, such as messaging and profiling; full availability of services.

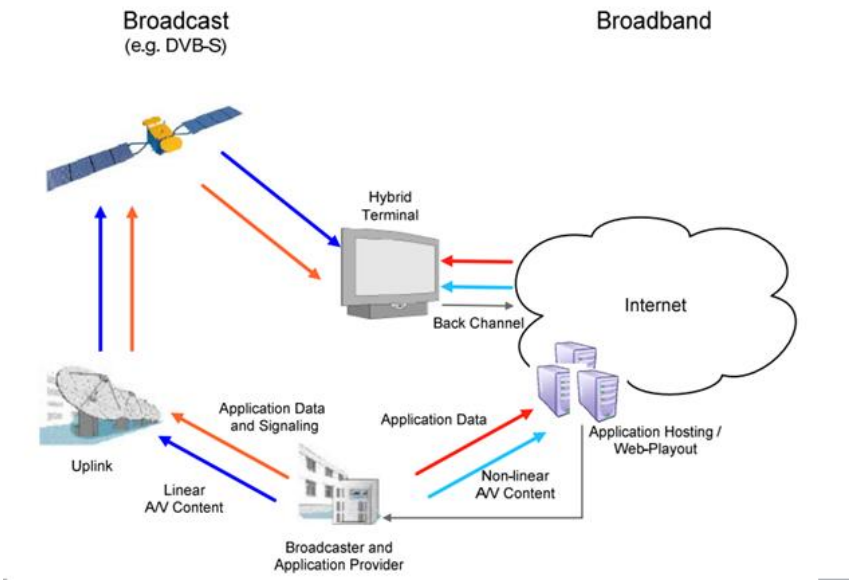

Figure 5: Interactive profile

- **Interactive with a Second-Screen** (user input via remote-control also possible)**:** broadcast and internet access; fully availability of services enhanced with a second-screen functionality for intuitive input.

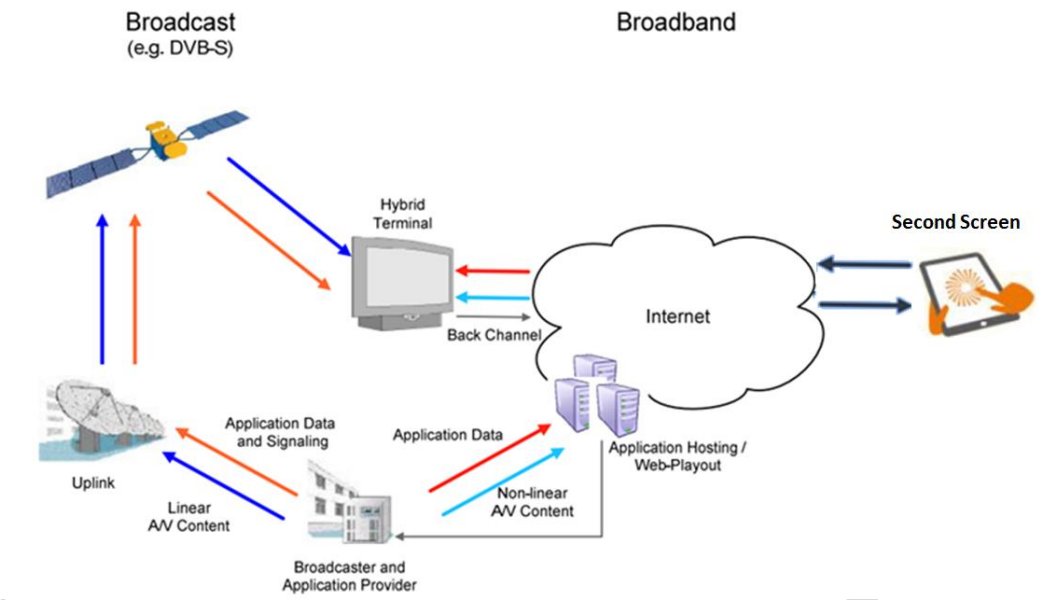

Figure 6: Interactive profile with Second-Screen

For the time being the further description focusses on the development of the “Interactive Profile” as a fundament for the other scenarios mentioned. These will be emerged and derived from this profile in the later process of the development.

4. PROTOTYPING

The following section describes the development of the GeTVivid TV client and its embedded scenario.

4.1 Scenario Implementation

As depicted in section **Fehler! Verweisquelle konnte nicht gefunden werden.** the integration of the TV client in the targeted infrastructure is a crucial point of the business plan. For the implementation of the prototype, it was decided that the application shall be accommodated in a set-top-box. In order to achieve this goal, collaboration with TARA Systems GmbH was established. The software company develops amongst others embedded software for set-top-boxes. Therefore an already existing prototype was adapted, being developed in an EU funded project called Global iTV (http://www.globalitv.eu/).

This STB (set-top-box) inherits a software stack, which supports the HbbTV standard and delivers additional features which were aligned for the goals of the GeTVivid project. A major obstacle for the project was that, due to the HbbTV standard, an application is bound to the broadcast stream and killed when the user switches the channel. The solution by TARA systems incorporated in their software stack was to provide an additional second HbbTV instance to the already existing one, which is globally available independently from the tuned channel against the constraints of the standard. This dedicated instance is solely for the purpose to host the GeTVivid application, which when active, overrides the “conventional” broadcast HbbTV application. Once it is closed, the initial broadcast carousel and application is restored. The application can also be activated after the system start-up and therefore is kept alive during the whole lifecycle. Furthermore it does not depend on the DVB signal; it can also use an IPTV stream and add the HbbTV functionality.

Figure 7: Tara Systems set-top-box prototype

As a result for the user-setup, only this modified STB is needed, which is directly connected to the internet and to the TV via HDMI or SCART, enabling older television sets without HbbTV support to be ready for the GeTVivid platform.

Another adaptation of the prototype is the implementation of a third “global” HbbTV browser instance for the GeTVivid notification feature, which allows displaying messages on the TV screen. This additional browser

provides an overlay screen to the broadcast programme in order to display messages in a pop-up-style manner when the user is watching TV and not interacting with the platform. This adaptation is an extension of the broadcast profile described in section **Fehler! Verweisquelle konnte nicht gefunden werden.**, as the user does not interact with the TV remote set, but still requires a broadband connection in order to actually display the notification messages.

4.2 Client Implementation

The implementation of the GeTVivid TV client (see Figure 7) is divided into two major parts: the frontend and the communication with the server. As the communication with the server is defined through interfaces on the server-side, the focus of the following sections is on the frontend, i.e. on the display of pages, the handling of user interaction as well as the logic combining the two. As suggested in the "10 golden rules for HbbTV app development" by the Fraunhofer Fokus institute (Kraus & Seeliger, 2014), the frontend implementation heavily makes use of existing design patterns to provide stable, high quality code and leverage its re-usability.

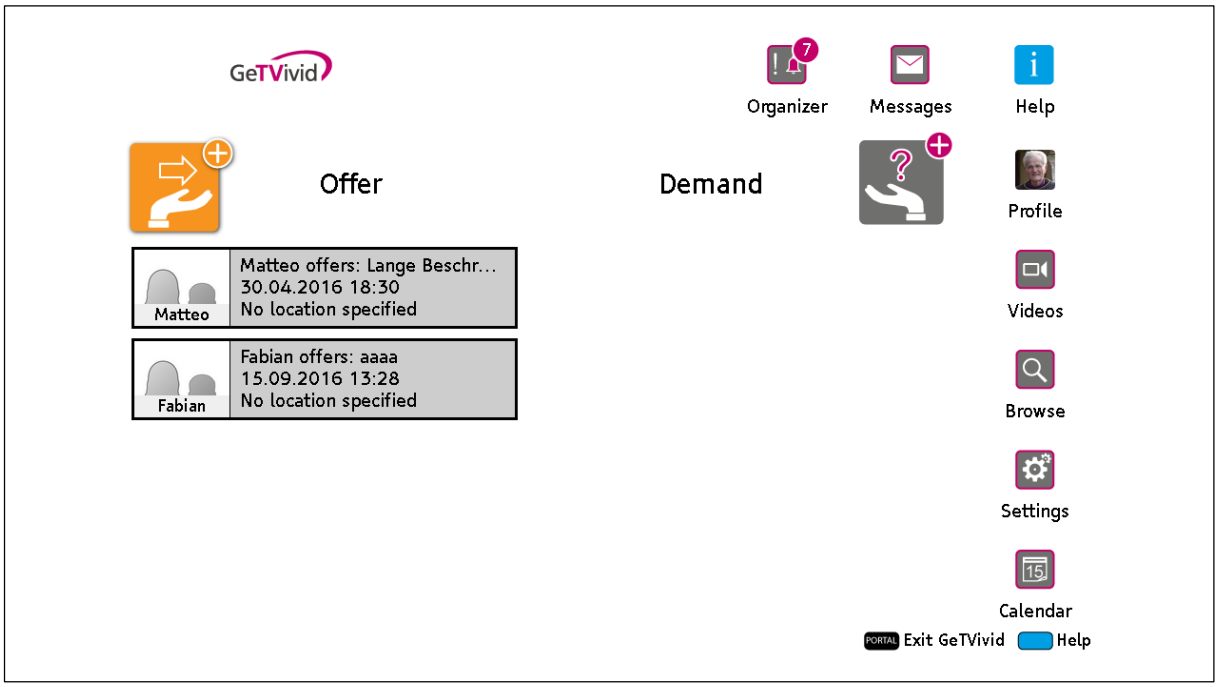

Figure 8: Main menu of the GeTVivid TV client

The following list provides a short overview of the patterns used in the TV client:

- Model-View-Controller: The idea of this pattern is to separate the model (underlying entity of an element), the view (UI element displayed to the user) and the controller (object used to communicate user input to the model). This allows manipulating the model in a unified way, as it is loosely coupled to the view, making it re-usable throughout the application.

- Façade: Complex processes that require a sequence of logical steps to be executed can be simplified using this pattern. It defines a commonly accessible object (the *Façade*) that executes these steps for clients through an easy-to-understand interface.
- Observer: The Observer pattern is used to provide an abstract event handling mechanism. Clients (the *Observers*) can register themselves at a commonly accessible subject, who will notify all registered clients on a state change.

In the remainder of this section the design choices of the GeTVivid TV client will be analysed and the corresponding implementation details will be specified.

4.2.1 Page Layout

This section describes the considerations which were made regarding the page layout and its concept for the actual implementation.

4.2.1.1 Design Choices

The GeTVivid TV client uses a page layout as suggested by Fraunhofer Fokus (Kraus & Seeliger, 2014), using a *Scene Manager* to distinguish between commonly shared elements between pages and the individual content of a page (see Figure 9). The benefits of this design choice are, amongst others, the possibility to control states of different scenes from the manager and the prevention of a flickering broadcast video when changing from one page to another. Moreover, users have common interface elements they can rely on for orientation, with the navigation across pages being located at the top of the page and information on possible remote control input being located at the bottom of the page.

Figure 9: Scene manager and scene content of the TV client

Using a scene manager with a fixed header and footer also complies with another suggestion from Fraunhofer Fokus, namely the *Ten-foot User Experience.* They recommend that HbbTV applications shall be designed in a

way that the *Overscan* area does not contain any UI elements, the *Action Safe* area contains shortcuts for navigation purposes and the *Title Safe* area contains the main user interface. Figure 10 illustrates these areas for the GeTVivid TV client, making clear that all of the aforementioned rules are satisfied.

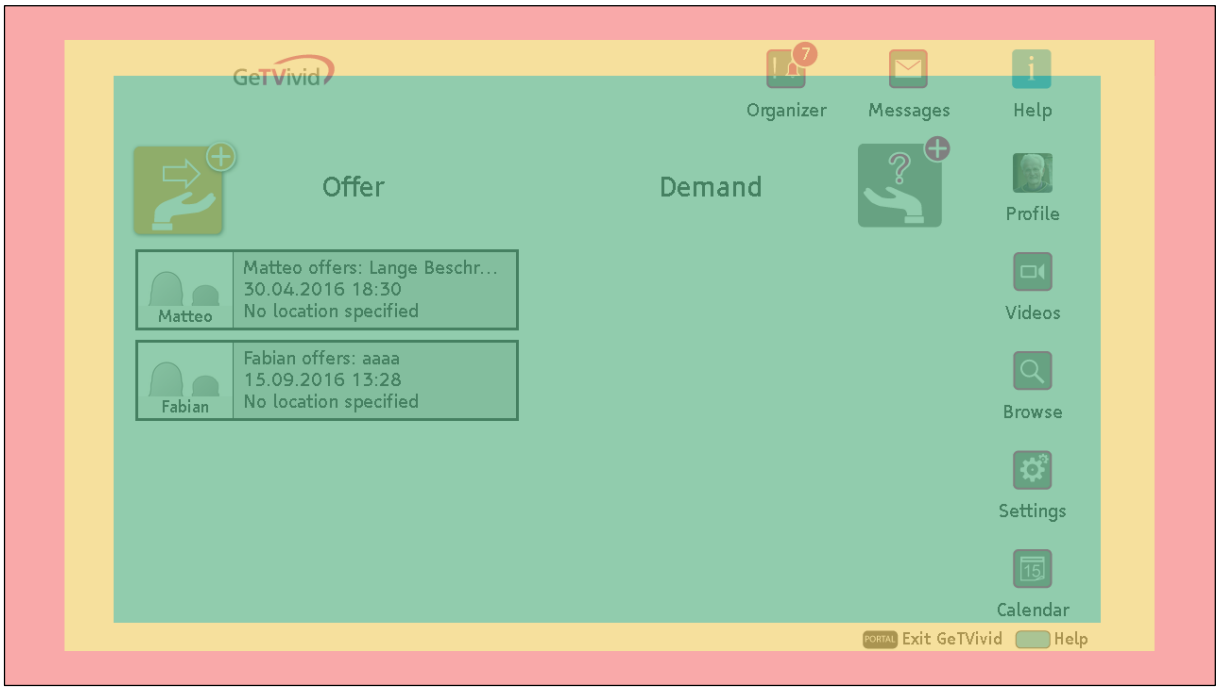

Figure 10: Overscan, Action Safe and Title Safe Areas of the TV client

4.2.1.2 Implementation

The scene manager is implemented as an object that is available to all pages (or *scenes*) of the GeTVivid TV client (see Figure 11). Each scene can configure the scene header by providing a header definition. The definition specifies, for example, how elements from the scene interact with elements in the header. Similarly, the scene footer can be configured using a definition that requests the set of buttons the user can use to interact with the application. Finally, scenes can add handlers to remote control keys in order to be notified once the user presses the corresponding key on the remote control. The implementation of this mechanism uses the observer pattern.

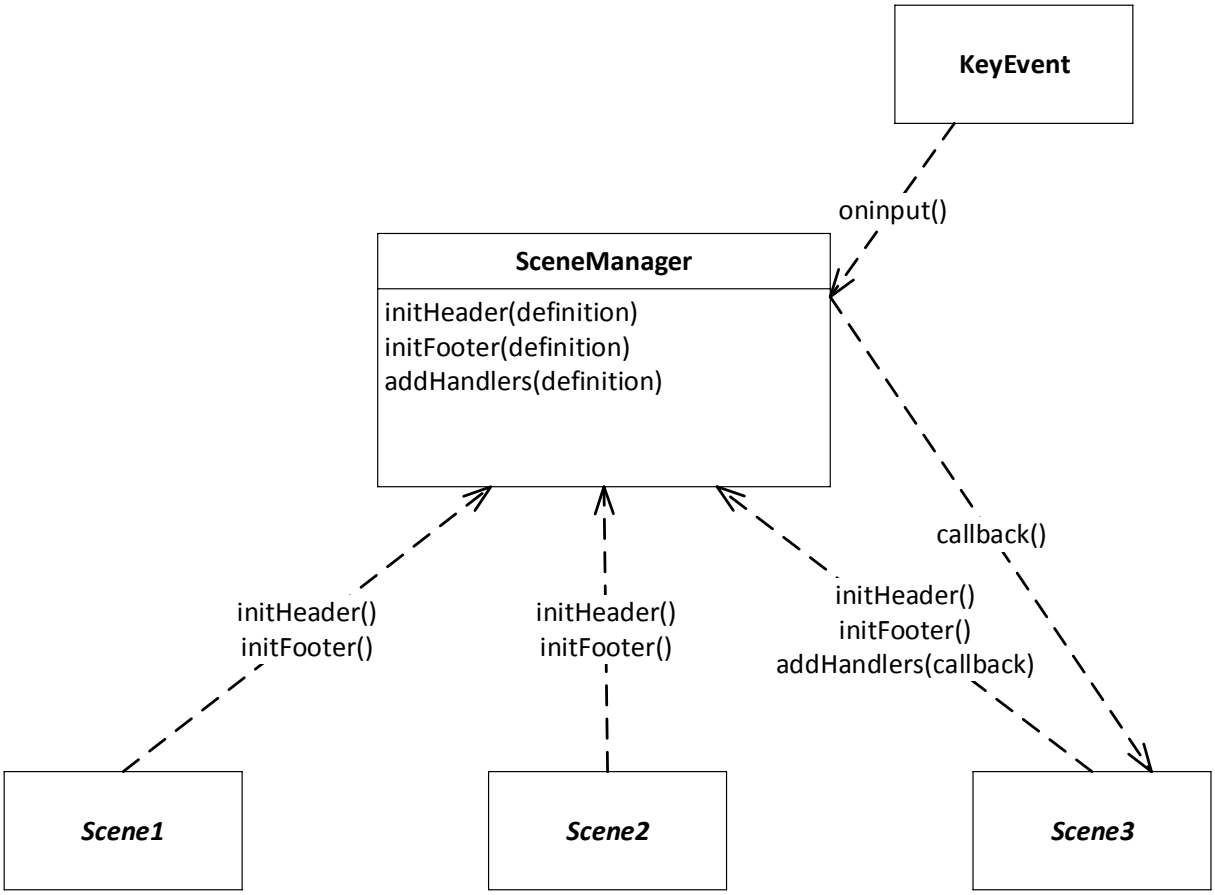

Figure 11: SceneManager implementation

4.2.2 Navigation

When dealing with interactive television, various issues have to be addressed when developing dedicated applications, due to the differences to commonly known web applications.

4.2.2.1 Design choices

Users of HbbTV applications do not have regular input devices such as mouse and keyboard available, but only a remote control. Therefore, navigation between elements of the application has to be considered carefully, so that

(a) every element of the page can be navigated to,
(b) it is clear for every element which actions are available,
(c) it is clear which element is currently focused/selected and
(d) it is clear to which elements the user can navigate from the currently focused element.

In (Kraus & Seeliger, 2014), any such element is referred to as a *TV optimized element*, also raising additional questions for each page:

(e) What kind of page am I visiting? What can I do?
(f) How can I go back / close the application?

All of these issues are addressed in the GeTVivid TV client using solutions that have been established either through the HbbTV developer community, the Web developer community or through official documents on the design of TV applications. In detail, the solution for each issue is implemented as follows:

(a) The navigation is implemented using the Document Object Model language binding for ECMAScript. Using this model, the navigation (UP/DOWN/LEFT/RIGHT) is explicitly defined for every element to ensure that every element can be navigated to. As different HbbTV devices use different mechanisms for navigation purposes, this is a reliable way to provide the same user experience across a wide variety of devices.
(b) Every focusable element in the GeTVivid TV client has an icon and two states: The default state and the focused state. In the default state the element will simply display its icon, while in the focused state it will display a highlighted icon. Every such element can also be selected using the *OK*-button of the remote control. If the element is a link to another page, as is the case in the menu page (see Figure 8), selecting the element will simply follow the link. If the element is selectable (e.g. an item from a list), the element has an additional state, namely the selected state. In this case, selecting the element will display the icon for the selected state (see Figure 12). Using this scheme, the actions for each element are clear (*OK*-button for every element that can be selected), where textual and visual hints help the user to distinguish between links and selectable elements. Any additional actions that are available throughout the app with no specific element associated with it are defined in the footer of the scene manager (see Page Layout).
(c) See (b) and Figure 12.

Figure 12: Default icon, focused icon and selected icon of a selectable GeTVivid UI element

(d) In order to assure that the user will always know which elements can be navigated to from the currently focused element, the elements are laid out in a matrix-like fashion where applicable (see Figure 8). The header of the scene manager is regarded as a separate part of the page's main content and is therefore not clearly aligned with the page's main element matrix.
(e) Every page of the GeTVivid TV client clearly displays the icon associated with the scene as well as the scene's title in order to provide information about the kind of page currently visited (see Figure 13). Furthermore, textual headers inform the user about the available actions that can be executed on the current page. As the same icons are used for similar actions and every selectable item has a label to indicate its actions, users can quickly identify what actions can be executed on which element.
(f) Meeting the HbbTV standards, the red button on the remote control can be used to close the application at any time. Once closed, the application can be re-opened by pressing the red button again. In the

application itself, there will be a *Back*-button in the top left corner of every page (except for the main menu) which can be used to go back within the application. As a shortcut, the user can also use the *Back*-button on the remote control for the same effect. In addition, the yellow button on the remote control can be pressed on any page in the application to navigate back to the main menu.

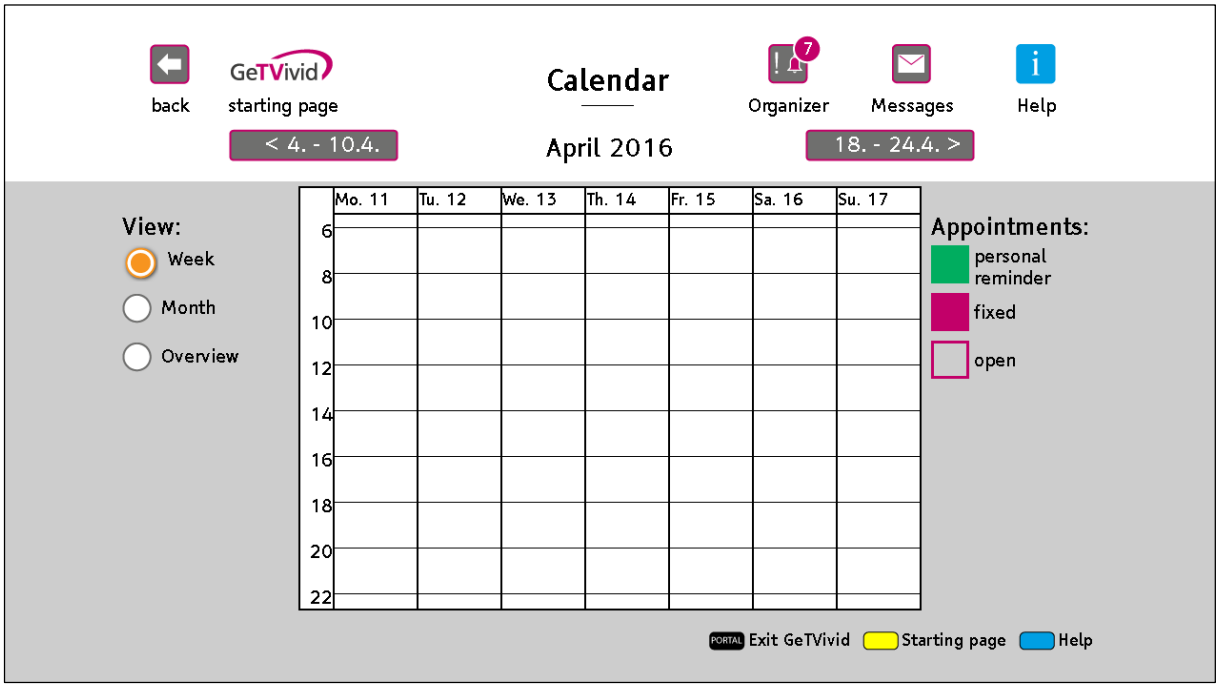

Figure 13: Calendar page of the GeTVivid TV client

4.2.2.2 Implementation

Both, the page icon and title as well as the *Back*-button of the scene header are realised through the *SceneManager* as illustrated in Figure 11, by configuring the header respectively. The usage of the colour buttons (RED to close application, YELLOW to navigate to the main menu) as well as the BACK button on the remote control is globally defined by the *SceneManager* as well. In a similar fashion, the navigation keys (UP/DOWN/LEFT/RIGHT) on the remote control will change the focused element, if an element has been defined for the requested direction of the currently focused element. Navigation directions for elements are specified whenever the layout of a page changes, for example, when a scene is loaded for the first time, when items are added or removed or if a message window is shown.

The title and icon of a scene are specified when the scene header is initialized through the *SceneManager* (see Figure 11). If such a definition is provided when initializing the scene header, the logo will align with the remaining header elements, indicating that it may be focused. In that case, users can select the logo at any time to return to the main menu of the application.

Due to the various icons that are used throughout the application, the loading time of a page can be a problem. Parts of the page may not be displayed, while interaction with some elements may not be possible until the page is fully loaded. In applications installed on a device, images are usually provided along with the installation files to speed up the loading process. As HbbTV applications are similar to web applications and cannot provide any installation files, all images have to be fetched remotely. Browsers will therefore request each image from a

remote (file) server. The data that has to be transmitted is then not only the image data itself, but also the client's request to the server as well as the server's response. As browsers will fetch each image separately, the request and response payload will be added for every image used on the page. A technique to circumvent this procedure is commonly known as *CSS Sprites*. The idea is to only provide a few (in our case two) large image files, which contain all images used on the page. Figure 14 illustrates how such an image file looks for the GeTVivid TV client. One major benefit is the drastically reduced number of server requests and responses that have to be transmitted – without CSS Sprites, the TV client would require about 100 requests and responses, with CSS sprites it is reduced to two. The other benefit is the simplicity with which the highlighting of elements can be executed when using this technique.

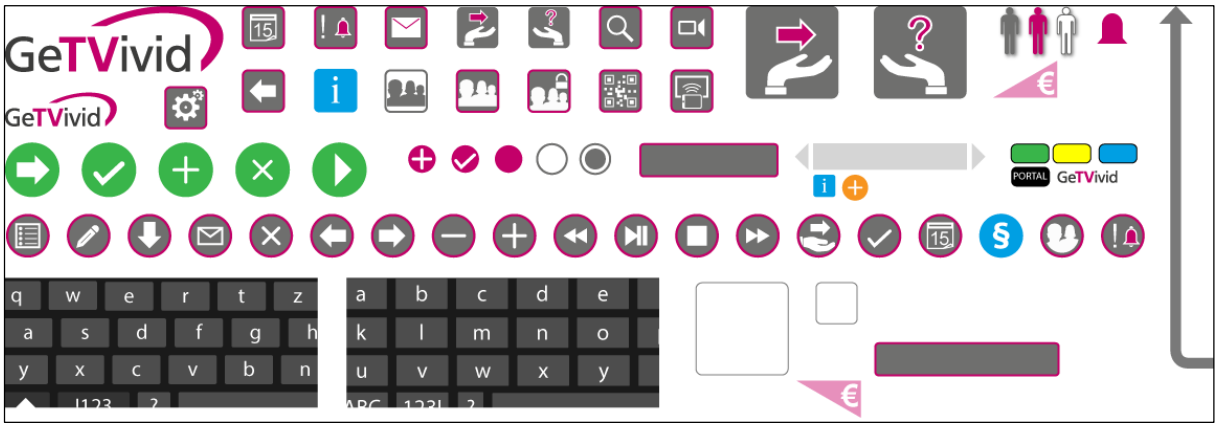

Figure 14: CSS Sprites image file containing icons of the TV client

Displaying an icon for an element basically requires three steps. First, the background of the image has to be set to the image file containing all icons. Afterwards, the position of the background has to be set to the location of the icon in the image file (e.g., background-position-x: 500px; background-position-y: 200px). Finally, the width and the height of the icon element have to match the icon in the image file, so that none of the adjacent icons is visible by accident. This very simple procedure is easily executable and maintainable – changing an icon simply requires replacing the icon in the CSS Sprite image file. Adding new icons is just as straightforward, as the image file can be extended at the bottom and at the right without interfering with the position of the original icons. In addition, as mentioned before, highlighting of icons is greatly facilitated, as neither the icon's position nor its size has to be updated. Instead, the CSS Sprite image file is replaced with a similar file containing only highlighted icons, whenever the element is focused.

4.2.3 Input

The issue of how to control the application will be a crucial factor for the acceptance of the service to the user. On the one hand older adults will be familiar with the concept of a remote control; on the other hand they may be not used to the basic navigation concepts known from web applications or mobile devices.

4.2.3.1 Design choices

As mentioned before, for the GeTVivid TV client the only guaranteed input device is the HbbTV device's remote control. However, the application requires textual input on some pages, which cannot be entered with the remote control only. Although the HbbTV specification states that every terminal shall provide a method to enter

text (Kraus & Seeliger, 2014), neither the appearance nor the mechanism (soft keyboard vs. *multi-tap*) is explicitly defined. Hence, the appearance or mechanism to input text will differ based on the HbbTV device the application is being executed on. This may result in problematic behaviour on some devices, as the provided input method may not be suitable for the target group of the GeTVivid project and the effect the input method may have on the appearance of the page cannot be controlled. For instance, a soft keyboard provided by the terminal may hide content that is relevant for the text that shall be entered. In order to address these issues, the TV client will provide its own soft keyboard implementation used to enter text (see Figure 15).

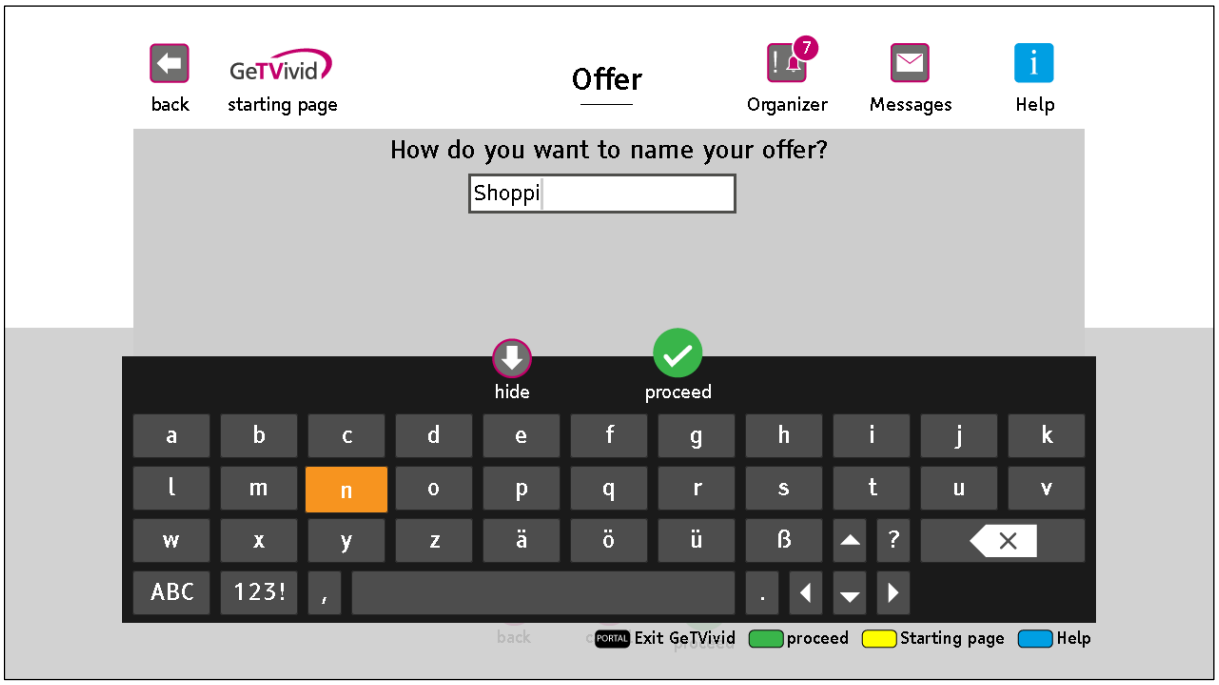

Figure 15: Soft keyboard used for text input

The keyboard elements follow the same rules as any other element of the GeTVivid TV client, providing a consistent user experience throughout the application, in contrast to terminal defined input methods, which are inconsistent with the rest of the application design. Another advantage of this input method specific to the GeTVivid project is the possibility to enlarge the input field, moving it to the centre of the screen and providing a header explicitly stating what the user is expected to enter. Numeric input is realized by displaying a numeric keypad that doesn't allow for letters to be entered (see Figure 16). Alternatively, numbers can be entered using the remote control's numeric keys (0-9).

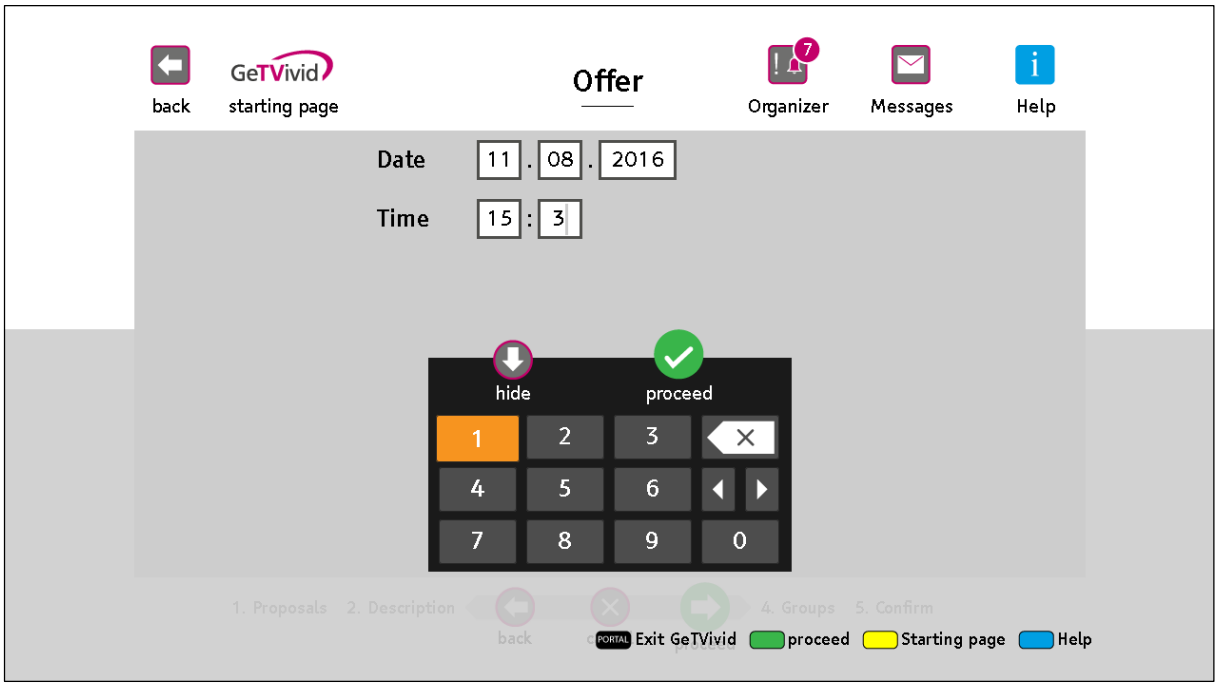

Figure 16: Soft keyboard used for numeric input

4.2.3.2 Implementation

The TV client's soft keyboard is implemented using the *Façade Pattern* (see Figure 17) in order to simplify the control over the input method, providing a more stable implementation of the application. In general, any scene in the application can register a UI element for input through the scene manager. The manager will then create a new input object and associate it with the provided element, returning a handle to the created object. Once registered, the input for this element (i.e. the soft keyboard) can be shown using the previously received handle. The user can then use the remote control to enter text through the soft keyboard, which is realized using the *add*/*remove* methods of the corresponding input object. If the user completes the input process, the input object will be finalized, giving control back to the scene that requested the input to be shown in the first place. Furthermore, scenes can define optional attributes of input objects using the corresponding input handle. This allows to define the maximum length of an input field (e.g. when entering a time) or the input field to proceed to once the user leaves the current input field. See Annex A for the source code of the input module.

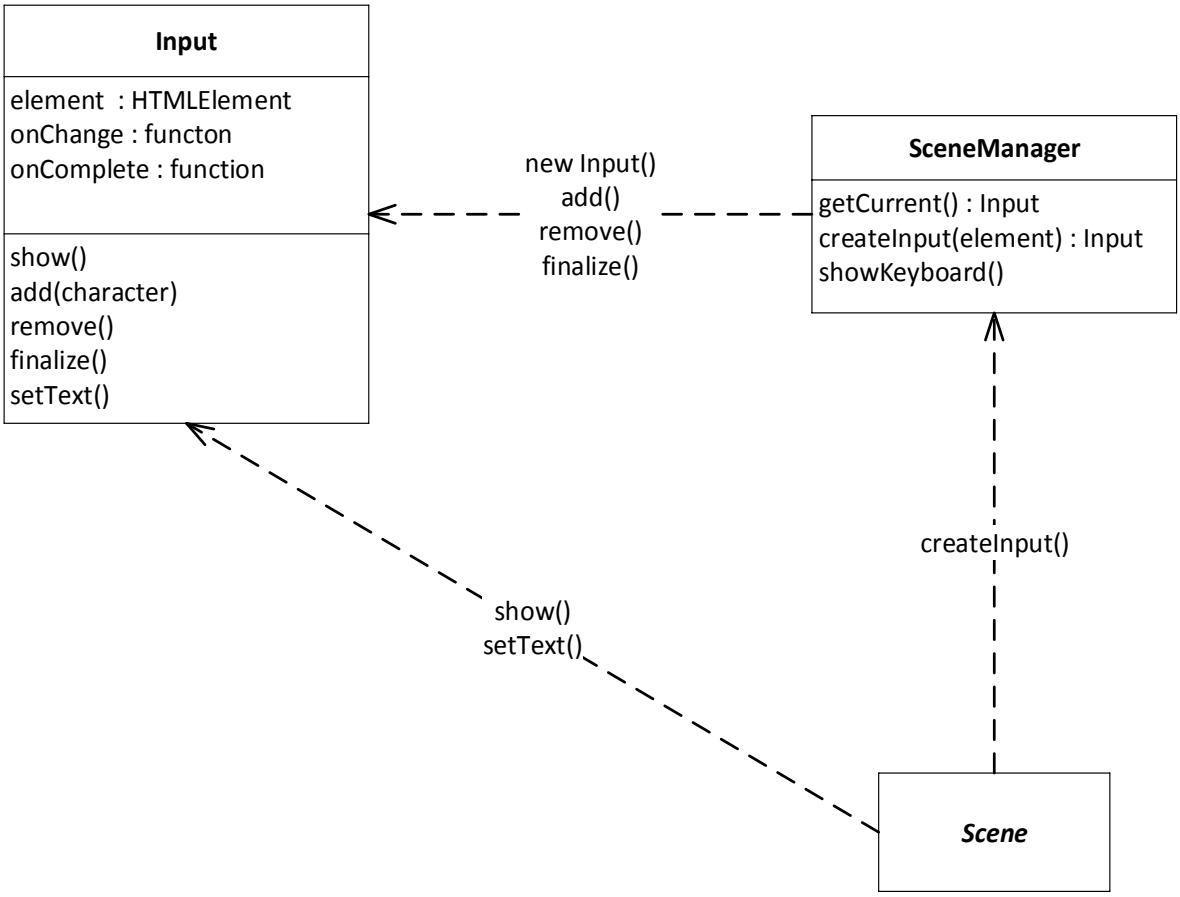

Figure 17: Input method implementation using facade pattern

4.2.4 Dialogs

The concept of dialogs is crucial across all kinds of applications, be it native, hybrid or web, desktop or mobile applications. They allow displaying additional information to the user, without interfering with the layout of the current page. Moreover, they force the user to focus on the piece of information the dialog is distributing, hindering them to take any action until the dialog is completed. Another important scenario is the confirmation of previously taken actions, in order to prevent an accidental selection of an element to have an impact on the usage of the application. This section describes how matched dialogs help to enhance the user experience for this application and how they are used throughout the system.

4.2.4.1 Design choices

Dialogs are a very useful tool in the application to provide additional information to the user as well as allowing the user to take additional actions on an element. However, not all dialogs are triggered by the user – for instance, a dialog might be shown if the user receives a notification (see Figure 18). As multiple dialogs being displayed at the same time are an issue that affect not only the layout and navigation of the page, but might also confuse the user, there will always be only one dialog at a time. For instance, if the user receives several notifications at once, only one dialog will be shown displaying information about the most recent notification.

Once this dialog is closed, information about the other notifications is shown in a separate dialog. Another important factor that reduces the confusion to a minimum is the fact, that all dialogs use the same basic layout:

- Dialog title and/or text are in the upper part of the dialog window
- Buttons used for additional actions are always in the lower part of the dialog window, where the “close”-button is always at the very left and is always focused initially
- If applicable, a large icon indicates the type of the dialog currently displayed
- Secondary text or icons are shown between the upper and lower part of the dialog window

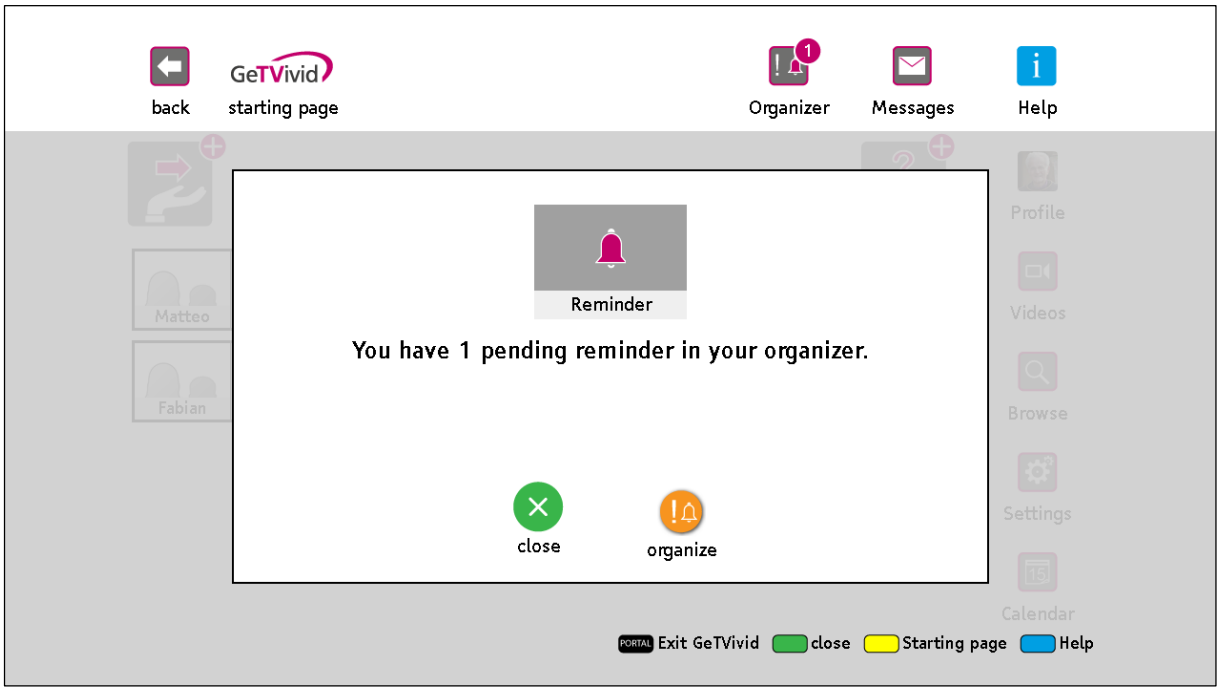

Figure 18: Notification dialog

There are four basic types of dialogs:

1. Confirmation dialogs, that allow the user to confirm or cancel an action (e.g., before an input is sent to the server). The only buttons present are the “close”-button and the “confirm”-button. The dialog can optionally be confirmed using the green button on the remote control.
2. Notification dialogs, used to display information received from the server. A notification dialog is displayed once a reminder defined by the user is within a certain time frame (e.g., two days before an appointment) or to inform the user about an upcoming event. Secondary notification dialogs only inform the user about pending notifications, without displaying any detailed information. Any secondary dialog has a “close”-button and a link to the notification page. Dialogs also displaying notification details have an additional link that will take the user directly to the corresponding calendar entry.
3. Detail dialogs (see Figure 19), which are only displayed once the user selects an item in the application. An offer in a list, for example, only shows the most important information like title, date and location. Once the user selects this offer, a detail dialog will be shown, providing additional information on the offer as well as on its creator. Moreover, the user can suggest a date for this offer or, if a date has been specified by the creator, accept the already specified date.

4. List dialogs, displaying a list of items from which the user can select. The dialog can be configured to only allow the selection of a single element, after which the dialog will be closed automatically, or to let the user select multiple items. In the latter case, the user has to close the dialog himself. The scene that opens a list dialog has to define the actions to be taken once the dialog is closed (automatically or manually).

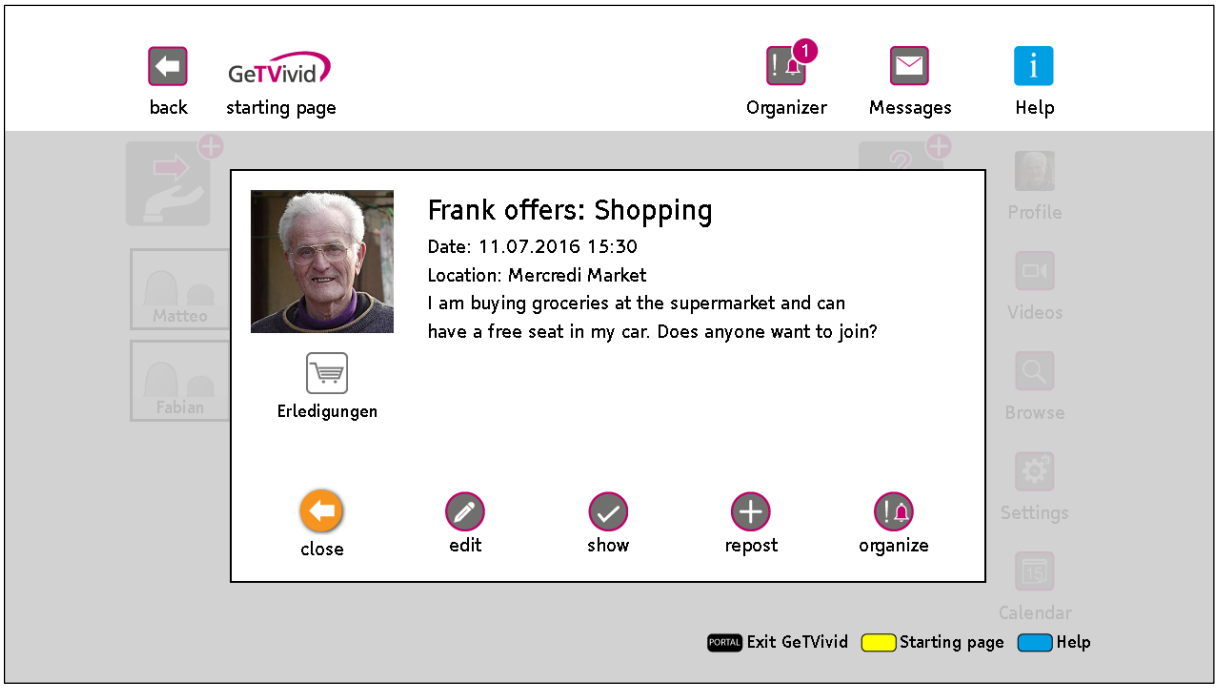

Figure 19: Detail dialog for an offer

4.2.4.2 Implementation

Similar to the input mechanism, dialogs are controlled entirely by the scene manager. Any scene can request a dialog to be displayed, causing the scene manager to take over control. At this point, the user can only interact with elements within the dialog; hence the focus can never leave the dialog window. If the user completes the dialog, the scene will regain control. Optionally, the scene can define the actions to be taken for the different outcomes of the dialog.

In the scenario of a confirmation dialog, the scene can define what should happen if the user cancels or confirms using the dialogs *onConfirmed* and *onCanceled* callbacks. Once the user takes one of these actions, the corresponding callback is executed, the dialog is hidden and the user can once again interact with all elements in the scene. An example of such a scenario is deleting a group. Once the user attempts to delete a group, a dialog is shown asking the user to confirm his action (see Figure 20). Closing the dialog will not execute any action in this case and take the user back from the dialog's "close"-button to the group. Confirming the dialog will remove the group locally and communicate the delete request to the server in the background. The dialog will be hidden and an appropriate element will be focused. Similar to the implementation of the Input module, the Dialog module is realised through the *Façade Pattern* (see Figure 21). See Annex B for the module's source code.

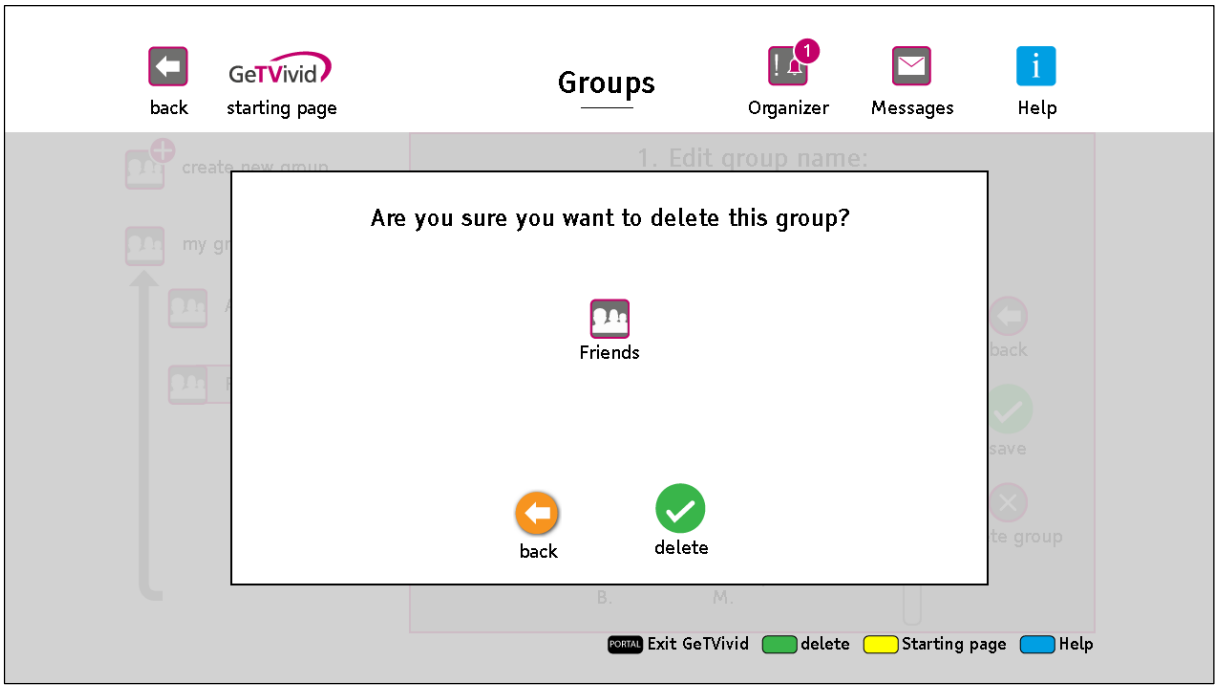

Figure 20: Dialog asking the user to confirm his action

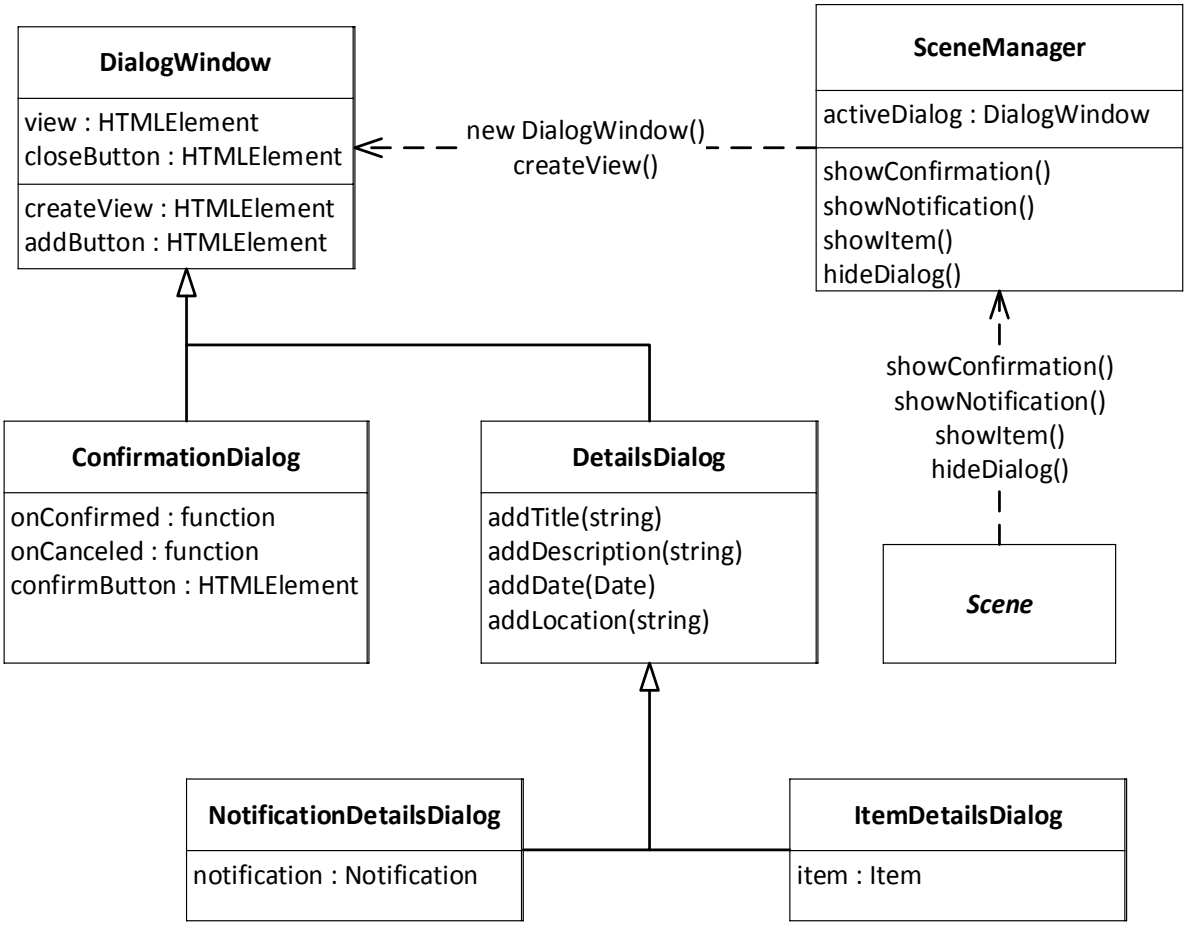

Figure 21: Implementation of the Dialog module

4.2.5 Lists

Whenever multiple elements are displayed next to or on top of each other, a list comes into play. Lists display several items at once and usually have additional items that are not directly visible, but can be viewed by scrolling through the list. On devices with more flexible input mechanisms like mouse and keyboard or touch input, scrolling through lists can be implemented easily. Scrolling on TV devices is more challenging if the elements within the list should be selectable (instead of just viewable, e.g., when reading a long text), as the arrow keys on the remote control are used to switch focus, and cannot be used directly to scroll the list. This section analyses the challenges when interacting with lists and presents the solution used for this TV application.

4.2.5.1 Design choices

There are several pages in the application where lists are required, usually to display existing offers and demands in the system. As the number of list items is basically limitless, lists can become an issue when it comes to performance and loading times, as the browser would have to initialize and display numerous items at once. Another concern already mentioned is the navigation within lists without confusing scroll mechanisms. Finally, it should be clearly indicated at which position the currently selected item is within in the list, e.g., by displaying a scroll bar.

HbbTV browsers in general usually address these issues and provide mechanisms to scroll through lists. However, the performance is usually a task the application developer has to take care of, the relation between navigation and scrolling is not consistent across browsers of different HbbTV devices and the appearance of the scroll bar cannot be defined and therefore differs when using the application on different devices. In order to provide a unique and consistent user experience on all HbbTV devices, the interaction with lists is controlled entirely by the application itself, leaving the browser only to display the items of the list in its current scroll state.

Although the scroll bars of scrollable elements cannot be defined in HbbTV browsers, it is possible to display elements that act like scroll bars. For this application, the scroll bar is a dark grey block inside a light grey block. The dark block moves whenever the scroll position of the scrollable element changes, in order to point out the current position within the element. If the list element cannot be scrolled (e.g., because all elements of the list are visible at all times), the scroll bar will not be shown at all. Figure 22 depicts the appearance of the application's scroll bar and how it will not be displayed if it is not required.

The navigation within lists is similar to the navigation between other elements throughout the application. The only difference is that the list will automatically be scrolled so that the selected element is entirely visible. If the element is already visible, the list will not be scrolled at all, otherwise the minimum amount of pixels required to display the element will be scrolled (see Figure 23). The currently selected element is highlighted by colour and increased size.

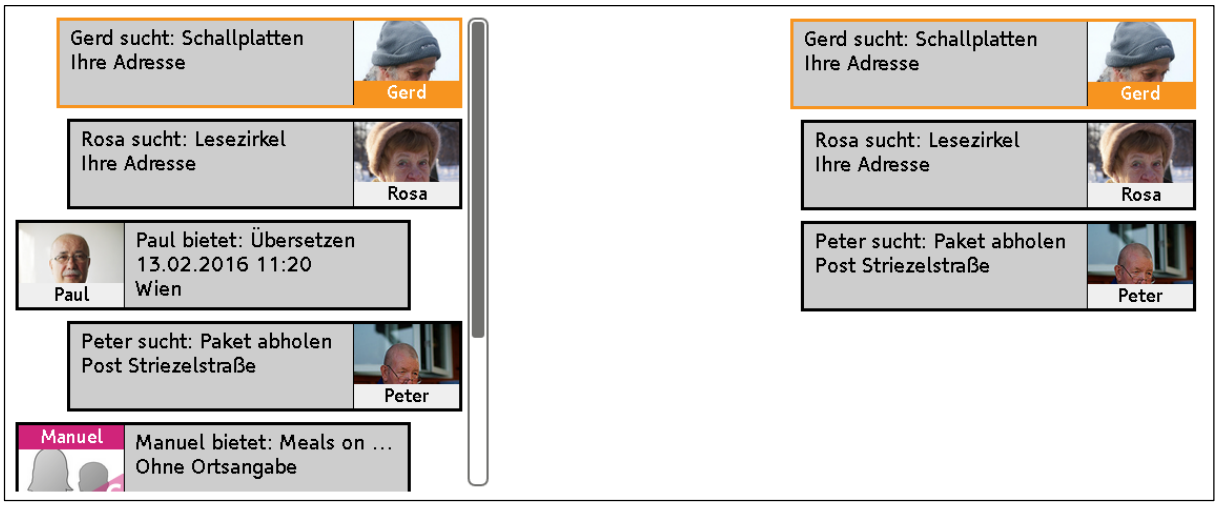

Figure 22: Scrollable list with and without a scrollbar

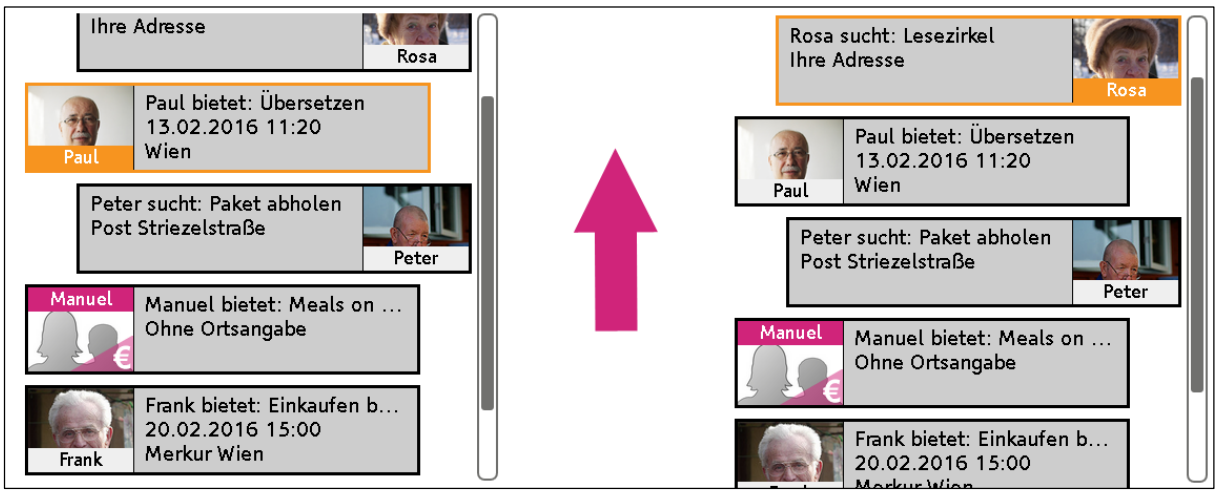

Figure 23: Scrollable list before and after scrolling up

In order to prevent lists with numerous items to slow down the page loading process, any scene will only load a small number of items when the page is loaded at first. If the user scrolls to the end of the list, additional items will be loaded and added to the list. The scroll bar will be adjusted automatically, indicating that additional items have been loaded. This procedure, often referred to as paging, allows to keep the loading time at a minimum and prevents unnecessary overhead, as only those items will be loaded that are actually required by the user. The items are usually sorted by relevance, so that e.g. the most recent items are shown at the beginning of the list. Older items, that are most likely of little interest to the user, are shown only if the user scrolls further down in the list. Using this mechanism, the number of items in lists does not have to be limited, so that users can access even the oldest items without any disadvantages regarding performance or page loading time.

4.2.5.2 Implementation

The scroll bar in lists is implemented using two HTML *div*-elements – one for the light scroll container and one for the dark scroll thumb. The elements will remain hidden as long as the list element's *scrollHeight* is less than the element's *clientHeight*, as the list cannot be scrolled in this case. The position of the scroll thumb within the container is computed using the list element's *scrollTop* relative to the list's *clientHeight.* The *scrollTop* property

will be updated when scrolling to a specific element within the list, which is realised by determining the index of the item within the list, and multiplying it with the element's height. Depending on whether the item is scrolled to from the bottom or from the top, the container is scrolled so that the item appears at the very top or the very bottom respectively.

The paging mechanism is implemented by enabling scenes to register a callback for when the list is scrolled to its very top (*onTopReached*) or its very bottom (*onBottomReached*). An example of this behaviour is the messenger, which allows users to send message to each other. If a user is selected in the messenger, the latest conversation will be loaded and its messages displayed to the user. In order to view older messages, the user can scroll up. However, not all conversations are loaded right away to increase performance, but only once the user scrolls to the end of the current conversation. Once that happens, the next messages in the conversation history will be loaded and displayed, as is depicted in Figure 24.

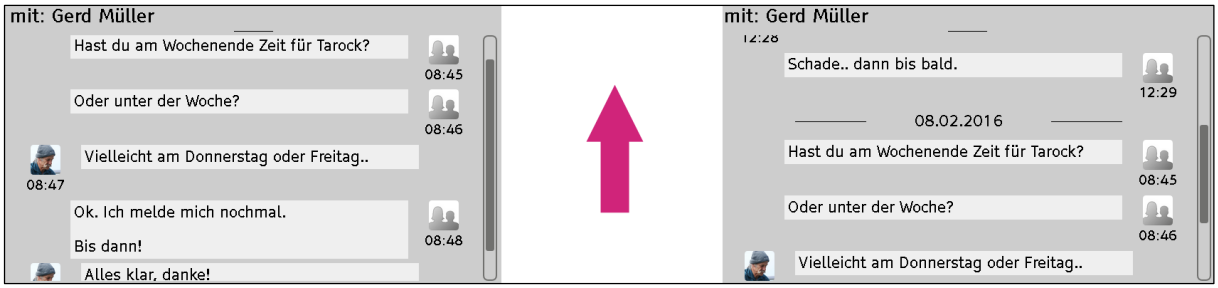

Figure 24: Messenger loading new message after *onTopReached* is executed

4.2.6 Second screen

Second screen functionality allows users to use a smartphone or tablet along with a TV set or computer as an additional screen to display information or as an alternative input mechanism. Several TV manufacturers provide native mobile applications that can be connected to a TV device and then be used as an advanced remote control for example. As text input is cumbersome using only the TV's remote control and navigation using a touch interface is sometimes more intuitive, this application provides second screen functionality as well. In this section the general purpose of the second screen as well as the implementation realising it are explained.

4.2.6.1 Design choices

When using a second screen users have to learn to use the application again on another device. Although some experience can be transferred, different input mechanisms change how the application is perceived and how certain actions can be taken. For instance, colour buttons on the remote control which are used as shortcuts in the application are not available on mobile devices. On the other hand, most of the shortcuts are irrelevant on mobile devices, as the corresponding elements can simply be clicked using the touch interface. Nevertheless, when designing the application for a second screen, the learning process of the user should not be neglected. Using the same layout on both screens and the same buttons with the same functionality for all actions, the required time to learn the second screen application can be reduced to a minimum. Furthermore, this design choice provides a consistent user experience, regardless of which device is being used.

Another important factor is the synchronisation between the devices. Although mobile devices allow using the application in an intuitive way, the TV device excels in displaying media information. Hence, users might switch

between devices while performing the same task, as some steps are easily executed on the mobile device, whereas others are better displayed on the TV. Therefore every step of each task should be synched between the devices, so that users are not limited to the device they started the task with. Lastly, users should be able to disconnect the devices again, so that they are no longer synchronised. At some point, a user might no longer wish to synchronise the devices, e.g. because another user is using one of the devices with another account.

4.2.6.2 Implementation

Connecting as well as synchronising the devices is greatly alleviated through the Second-Screen Framework (SSF) (Institut für Rundfunktechnik GmbH, 2014). This framework handles connecting the devices, maintaining the connection as well as exchanging messages between the devices. In order to connect the devices, the TV device has to initiate the connection process through the SSF, which will cause a QR-Code as well as an ID to be displayed on the TV screen (see Figure 25). The mobile device simply has to scan the QR-Code to connect the devices. Alternatively, the displayed ID can be entered on the mobile device if no camera is available.

Figure 25: Second Screen Framework connection manager

Once the devices are connected, any step taken in one of the clients is sent as a message to the other client, in order to keep the applications synchronised on both devices. The application defines a set of messages that can be exchanged, and each message is handled by the device that received it in order to adjust the layout and state of the page accordingly. Messages in the application consist of a type and a value. While the type is one of a few defined enumeration values (namely PATH, INPUT and SELECT), the message value can be anything from a string to any arbitrary object. As the SSF only supports strings to be sent as messages, any message value that is not a string will be converted to a JSON string first. Once the message is received by the other device, the JSON object will be parsed from the string received from the SSF.

PATH messages are sent whenever the scene (i.e. the path of the page displayed) changes, e.g. because the user clicked on a link. The value of such a message is the relative path to the page, which will be loaded as soon as the connected device receives the message.

Example value: offer.html

INPUT messages are exchanged whenever the user enters text on one of the devices. This text will be forwarded to the connected device, so that the input field displays the text originally entered. The value is an object with an ID property, which uniquely identifies the input field to synchronised, as well as the text that was entered.

Example value: { id : "offer-search", text : "Gardening" }

Finally, SELECT messages are sent for all other synchronisation required. Selecting an element that is not a link will send a synchronisation message for exactly that element. The connected device that receives a SELECT message can retrieve the corresponding element and act as if it were selected on the device itself. Any such message is an object consisting of a class and an ID. Both these values enable the receiving device to uniquely identify elements even across scenes.

Example value: { className : "Offer", id : "3" }

This structure is realised through a dedicated synchronisation module. Through this module any scene can send synchronisation messages, add listeners for incoming messages (*Observer Pattern*) and register elements that can be selected. Any registered element will automatically be selected whenever a SELECT message for that element is received. Figure 26 illustrates the implementation of the synchronisation module. The diagram includes an exemplary sequence of steps that would lead to the synchronisation of the devices.

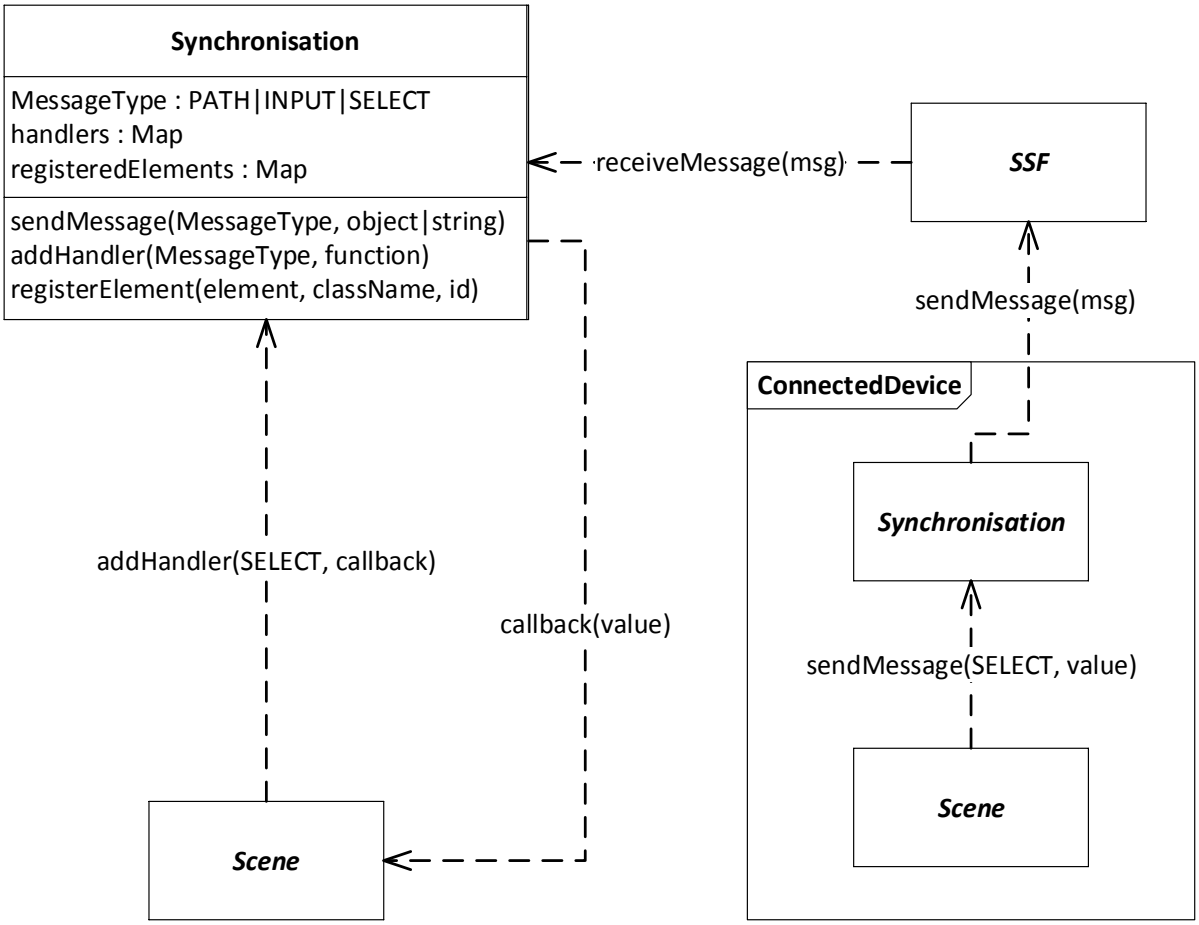

Figure 26: Implementation of the synchronisation module (using SELECT as an example)

4.2.7 Video player

As many of the target users are not accustomed to use applications on the TV via the remote control, a set of tutorial videos has been put together. These tutorial videos were made available through the client itself using a video player that is able to play back said videos. The design and implementation of the video player will be described in this section.

4.2.7.1 Design choices

In order to make the tutorials accessible to the user, the set of available videos needs to be displayed in some fashion to the user. For a clear and intuitive selection process, the videos are laid out in a matrix-like style, making it easy to select each video using the remote control's navigational keys. The matrix is depicted in Figure 27. Upon selecting a video, it will be opened in the client's video player, providing control mechanisms over the video itself as well as the possibility to close the player altogether.

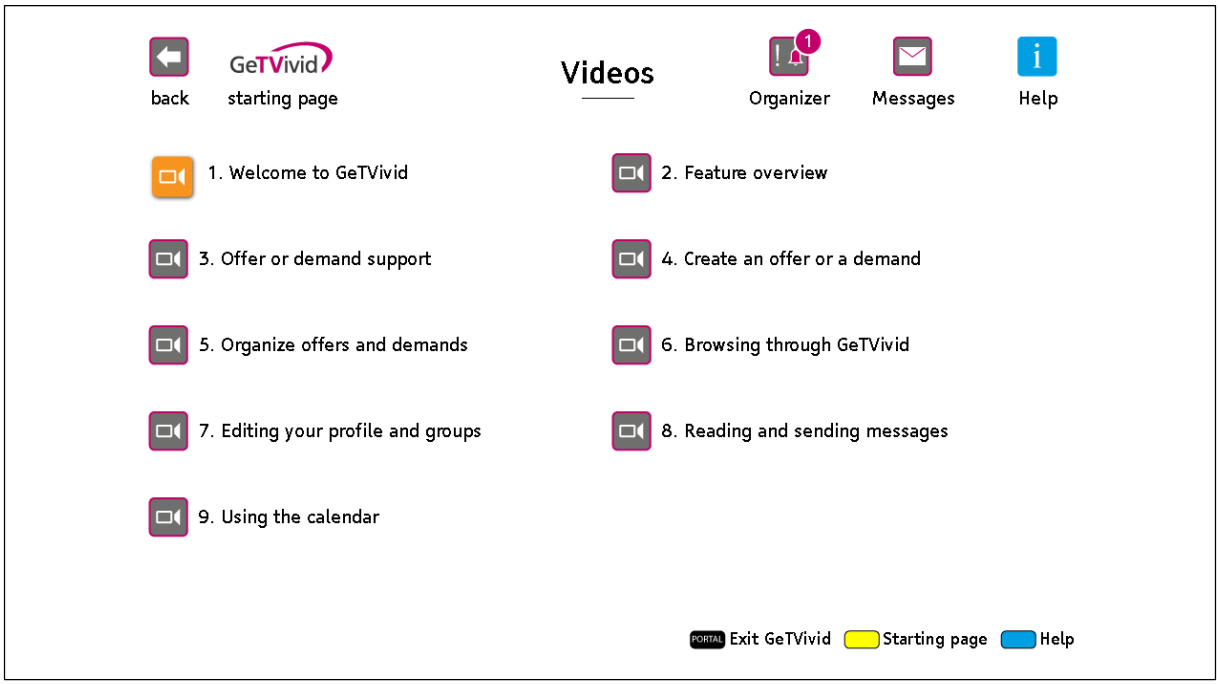

Figure 27: List of available tutorial videos

These control mechanisms are made available through the remote control's player buttons (play/pause/stop/seek) as well as on-screen buttons that can be selected using the navigational keys. This allows users who are already familiar with the way remote control keys can be used as a shortcut to have a smoother experience, but still provides an intuitive control for users who prefer to stick to a small set of keys on the remote control.

4.2.7.2 Implementation

The video list matrix was implemented by parsing a list of videos available on the server and setting up the navigation between them. The video list itself can be considered a linked list, in that each video knows its predecessor and successor, in order to allow the user to proceed to the next or go back to the previous video respectively. Selecting an item from the video matrix will open the video player and play back the video that is associated with the item.

As pointed out before, the video player is accompanied by a set of control buttons that are available directly below the actual video, providing the familiar play/pause/stop buttons as well as buttons to go to the next/previous video (see Figure 28). Once the video is finished, the user may select to replay the video, proceed to the next video or again view the previous video.

Figure 28: Video being played back in the player

The implementation of the player itself relies heavily on the HbbTV standard, which defines how videos should be initialized and controlled. Although the standard covers most of the functionality, some parts are left open to device specific implementations, which require specific adaptations in the client application. For instance, the space in which the video shall be displayed will be rendered black by some HbbTV devices, unless the video is actually being played back. That means that there will be a big black box where the video is supposed to be while the connection to the media server is established, during buffering as well as once the video has finished or was being stopped manually.

Another issue is that some devices do not fire all events as expected, in particular the event that should be fired once a seek operation has been completed (*onPlayPositionChanged*). On the other hand, some devices fire the event every second, making it impossible to rely on this specific event. Therefore, a custom listener that checks for a successful update for the play position was required. This listener checks the play position periodically until the position was updated correctly. Only then will the listener remove itself and the video will continue playing.

As the black box would disturb the user interface severely, an overlay was implemented that would hide the black box whenever the video wasn't playing. Once the video is stopped, finished or not ready to play yet, the overlay is shown, if the video is playing or being paused, the overlay is hidden. As the screen should not be entirely white while the video is not being played back, a loading symbol will be shown on top of the overlay whenever the video is being buffered or stopped. In case the video is finished, the title of the next video as well as the previous video's title will be shown instead, allowing the user to select one of the videos. The entire lifecycle of the video player and its relation to the overlay is illustrated in Figure 29.

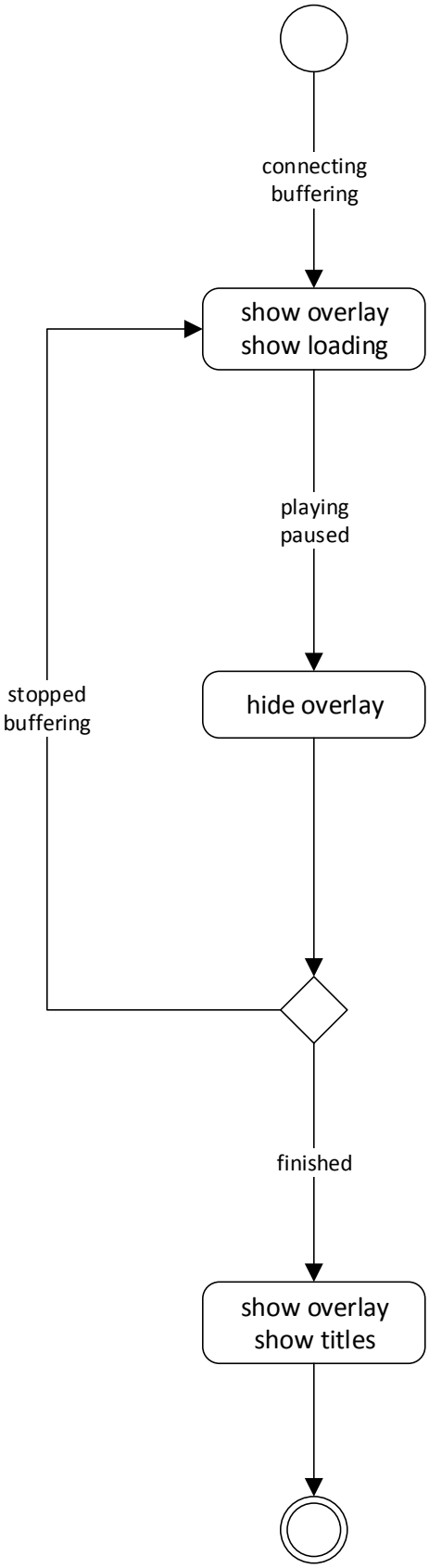

Figure 29: Video player activity diagram

4.2.8 Notifications

The GeTVivid backend provides a functionality that will inform the user about ongoing actions on the platform. For instance, whenever the user receives a new message, the server will inform the user immediately so that he can check his message inbox. The same holds true for changes in the user's appointments, e.g. when a user changes the date of an appointment. Obviously, these messages can only be received, if there is a connection to the server. As the TV client is a full screen application, the user will most likely only open the application if he is currently active, and not paying attention to updates on his activities. Therefore, a solution was implemented that allows the user to watch TV or even control other HbbTV applications and still receive server updates through on-screen notifications.

4.2.8.1 Design choices

As mentioned before, the notifications should be available both while watching TV and being active in other HbbTV applications. This implicitly requires on-screen notifications to be without user-input, because otherwise the input from the running application might interfere with the notification and the other way round. Hence, the notifications need to be self-dismissing (i.e. disappearing after a couple of seconds) and provide all necessary information without any input from the user.

However, the information display is limited in terms of space, as the user might be watching a TV show. The notifications should use only as much space as needed and as little space as possible, to prevent disturbing the user's viewing experience. In accordance with the de-facto standard approach to always display HbbTV popups in the bottom right corner, the on-screen notifications always appear in the same location. Moreover, the notifications make use of the horizontal space and hardly grow vertically, as the bottom of the TV image usually does not contain crucial information on the show that is currently running.

4.2.8.2 Implementation

The on-screen notifications that are displayed during broadcast or while using other HbbTV applications require a global browser instance that runs independently from all other applications. This global instance was provided, as mentioned in section **Fehler! Verweisquelle konnte nicht gefunden werden.**, by Tara Systems GmbH in an HbbTV set-top-box prototype. The prototype can be configured to always run an application, regardless of the current broadcast state. For the notification feature, a small application that only listens for server updates and displays notification popups was implemented.

On start-up, the application will establish a connection and immediately display any pending updates as on-screen notifications to the user (see Figure 30). In order to prevent notifications overlaying one another, only one notification will be shown at a time. If there are multiple updates, a queue with updates will be created, showing each notification only after the previous one has disappeared already. There are four types of notifications, each associated to a custom icon, allowing the user to quickly identify the notification type:

- Reminder notifications: Notifications about reminders the user has set in his calendar.
- Appointment notifications: Whenever changes to an appointment have been made, an appointment notification will be sent.
- Message notifications: Once the user receives a new message, a notification will be displayed.
- Award notifications: Any award the user receives will be sent as a notification to the user.

Figure 30: On-screen notification during broadcast

The application will only stop to listen for updates from the server once the TV client application is started, as the updates are directly available through the client itself, providing even more detailed information than is possible through on-screen notifications. Once the client application is closed, the notification cycle will continue after a short delay, fetching the latest updates from the server. As long as the client application is inactive, the notification application will never stop listening for updates from the server.

5. OVERALL CONCLUSION

As assumed from the beginning the usability seems to be a major factor for the success of the acceptance to the user. Due to a re-design of the overall user-interface, the TV client UI was reworked in multiple iterations.

Besides from the handling of the user interface, a first interoperability test of the TV client application within the scope of an HbbTV consortium workshop has been conducted. The application was tested on various current state and prototype TV devices of the leading manufacturers (e.g., Samsung and LG). There were only minor issues when referring to the functionality of the application, which have been addressed in the development process.

The original prototype offered the following functionalities:

- The TV client frontend according to the system and user requirements
- The relevant APIs to the GeTVivid backend, including the integration of other external system components (ACS Amiona, Profiling and Identity Management System)
- A messaging functionality including group management

A second version of the prototype has been developed for the lab studies, in which the prototype was tested with end users for the first time. In addition to the already mentioned functionalities, the following features are available for this prototype:

- Input and dialog mechanisms
- Scrollable lists
- Second screen functionality via the Second-Screen Framework

The final version of the TV client prototype was eventually developed to be tested with end users in the field trials of the project. In this version the already existing features were further enhanced and a few additional features have been added, namely

- Video player for tutorial videos
- On-screen notifications during broadcast

Being a prototype, the TV client can still be enhanced in several ways. In general, additional functionality of the system can be integrated through additional pages, but the upcoming HbbTV 2.0 specification provides further options to improve the user experience, e.g.,

- Companion screen support, allowing a smoother experience for second screen communication, also making it possible to launch the GeTVivid application on the TV from a mobile device (without using the set-top-box prototype),
- CSS animations, for a better visual experience, as elements can be altered visually using animations, instead of statically moving them from one state to another or
- Downloadable Fonts, providing a clearer view and style for different elements and pages of the application.

6. LITERATURVERZEICHNIS

ETSI. (2012). *Technical Specification 102 796 V1.2.1.* Sophia Antipolis: ETSI.

Institut für Rundfunktechnik GmbH. (26th. August 2014). HbbTV and Second Screen. *Digital Television And Online*. Abgerufen am 10. Juli 2015 von http://lab.mediafi.org/discover-secondscreenframework.html

K. Merkel. (2014). *Intensive Seminar on HbbTV.* Munich: IRT.

Kraus, C., & Seeliger, R. (2014). *10 Golden rules for HbbTV App development.* Berlin: Fraunhofer institute for open communication systems.

7. ANNEX A

```
var current = null;
var inputObjects = {};

function create(inputElement, onChange, onComplete) {
	var input;
	if(getvivid.util.hasClass(inputElement, "input-multi")) {
		input = new MultilineInput(inputElement, onChange, onComplete);
	} else if(getvivid.util.hasClass(inputElement, "input-numeric")) {
		input = new NumericInput(inputElement, onChange, onComplete);
	} else {
		input = new SinglelineInput(inputElement, onChange, onComplete);
	}
	inputObjects[inputElement.id] = input;
	return input;
}

function getCurrent() {
	return current;
}

function getHandle(id) {
	if(id in inputObjects) {
		return inputObjects[id];
	} else {
		return null;
	}
}


function Input(inputElement, onChange, onComplete) {
	this.inputElement = inputElement;
	this.onChange = onChange;
	this.onComplete = onComplete;
}

Input.prototype.show = function(suppressFocus) {
	this.addCursor();
	this.showKeyboard(suppressFocus);
};

Input.prototype.finalize = function() {
	var text = this.getText();
	if(typeof this.onComplete === "function") {
		this.onComplete(text);
	}
	getvivid.keyboard.hide();
	getvivid.hbbtv.focus(this.inputElement);
	current = null;
};
```

```
function SinglelineInput(inputElement, onChange, onComplete) {
	Input.call(this, inputElement, onChange, onComplete);
	this.maxLength = null;
	this.successor = null;
	this.predecessor = null;
}

SinglelineInput.prototype = new Input();

SinglelineInput.prototype.add = function(text) {
	this.inputElement.value += text;
	if(this.maxLength && this.inputElement.value.length >= this.maxLength) {
		if(this.successor) {
			this.proceedTo(this.successor);
		} else {
			this.finalize();
		}
	}
};

SinglelineInput.prototype.remove = function() {
	if(this.inputElement.value.length > 0) {
		var newLength = this.inputElement.value.length - 1;
		this.inputElement.value	=	this.inputElement.value.substring(0,
newLength);
	}
};

SinglelineInput.prototype.getText = function(text) {
	return this.inputElement.value = text;
};

SinglelineInput.prototype.setText = function(text) {
	this.inputElement.value = text;
	if(typeof this.onChange === "function") {
		this.onChange(text);
	}
};

SinglelineInput.prototype.setSuccessor = function(successor) {
	this.successor = successor;
	if(successor) {
		successor.predecessor = this;
	}
};

SinglelineInput.prototype.proceedTo = function(input) {
	this.finalize();
	input.show(true);
};

SinglelineInput.prototype.showKeyboard = function() {
	getvivid.keyboard.showSinglelineLetterKeyboard();
};
```

```
function MultilineInput(inputElement, onChange, onComplete) {
	Input.call(this, inputElement, onChange, onComplete);
	this.lineIndex = 0;
	this.nodes = [];
}

MultilineInput.prototype = new Input();

MultilineInput.prototype.add = function(text) {
	this.nodes[this.lineIndex].nodeValue += text;
};

MultilineInput.prototype.remove = function() {
	var node = this.nodes[this.lineIndex];
	if(node.nodeValue.length > 0) {
		var newLength = node.nodeValue.length - 1;
		node.nodeValue = node.nodeValue.substring(0, newLength);
		this.cursorIndex--;
	} else if(this.lineIndex > 0) {
		this.inputElement.removeChild(node.previousSibling);
		this.inputElement.removeChild(node);
		this.nodes.splice(this.lineIndex, 1);
		this.lineIndex--;
	}
};

MultilineInput.prototype.getText = function() {
	var text = "";
	for(var i = 0; i < this.nodes.length; i++) {
		if(i > 0) {
			text += "\n";
		}
		text += this.nodes[i].nodeValue;
	}
	return text;
};

MultilineInput.prototype.setText = function(text) {
	var textElements = convertToHtml(text);
	this.nodes = [];
	removeAllChildren(this.inputElement);
	for(var i = 0; i < textElements.length; i++) {
		var textNode = textElements[i];
		this.inputElement.appendChild(textNode);
		if(i % 2 === 0) {
			this.nodes.push(textNode);
		}
	}
	this.lineIndex = this.nodes.length - 1;
	if(typeof this.onChange === "function") {
		this.onChange(text);
	}
};

MultilineInput.prototype.showKeyboard = function(suppressFocus) {
	getvivid.keyboard.showMultilineLetterKeyboard();
};
```

```
function NumericInput(inputElement, onChange, onComplete) {
	SinglelineInput.call(this, inputElement, onChange, onComplete);
	if(typeof onChange === "function") {
		this.onChange = function(text) {
			onChange(parse(text));
		};
	}
	if(typeof onComplete === "function") {
		this.onComplete = function(text, isIntermediate) {
			onComplete(parse(text), isIntermediate);
		};
	}
	function parse(text) {
		if(text && !isNaN(text)) {
			return parseInt(text, 10);
		} else {
			return null;
		}
	}
}

NumericInput.prototype = new SinglelineInput();

NumericInput.prototype.showKeyboard = function() {
	getvivid.keyboard.showNumberKeyboard();
};
```

8. ANNEX B

```
var activeDialog = null;
var dialogDisplay = document.getElementById("dialog-display");

function showDialog(dialogWindow) {
	hideDialog();
	dialogDisplay.appendChild(dialogWindow.createView());
	show(dialogDisplay);
	focus(dialogWindow.closeButton);
	activeDialog = dialogWindow;
}

function hideDialog() {
	if(activeDialog !== null) {
		dialogDisplay.removeChild(activeDialog.view);
		hide(dialogDisplay);
		activeDialog = null;
	}
}

function showConfirmation(messages, onConfirmed, onCanceled) {
	showDialog(new ConfirmDialog(messages, onConfirmed, onCanceled);
}

function showItem(item) {
	showDialog(new ItemDetailsDialog(item), false);
}

function showNotifications(notifications) {
	var length = notifications.length;
	if(length > 0) {
		var dialogWindow = new NotificationDetailsDialog(notifications[0]);
		showDialog(dialogWindow);
	}
}
```

```
function DialogWindow() {
	this.view = null;
	this.closeButton = null;
}

DialogWindow.prototype.createView = function() {
	this.view = document.createElement("div");
	this.closeButton = this.addButton("close", hideDialog);
	return this.view;
};

DialogWindow.prototype.addButton = function(label, onClick) {
	var buttonElement = document.createElement("button");
	var buttonIcon = document.createElement("span");
	buttonElement.appendChild(buttonIcon);
	buttonElement.appendChild(document.createTextNode(label))
	buttonElement.onclick = onClick;
	this.view.appendChild(buttonElement);
	return buttonElement;
};

DialogWindow.prototype.addLink = function(label, url) {
	var linkElement = document.createElement("a");
	var linkIcon = document.createElement("span");
	linkElement.appendChild(buttonIcon);
	linkElement.appendChild(document.createTextNode(label))
	linkElement.href = url;
	this.view.appendChild(linkElement);
	return linkElement;
};

function MessageDialog(messages) {
	this.messages = messages;
}

MessageDialog.prototype = new DialogWindow();

MessageDialog.prototype.createView = function() {
	var view = DialogWindow.prototype.createView.call(this);
	for(var i = 0; i < this.messages.length; i++) {
		var messageParagraph = document.createElement("p");
		var messageText = document.createTextNode(this.messages[i]);
		messageParagraph.appendChild(messageText);
		this.view.appendChild(messageParagraph);
	}
	return view;
};
```

```
function ConfirmDialog(messages, onConfirmed, onCanceled) {
	MessageDialog.call(this, messages);
	this.onConfirmed = onConfirmed;
	this.onCanceled = onCanceled;
	this.confirmButton = null;
}

ConfirmDialog.prototype = new MessageDialog();

ConfirmDialog.prototype.createView = function() {
	var view = MessageDialog.prototype.createView.call(this);
	var icon;
	if(typeof this.onCanceled === "function") {
		this.closeButton.onclick = this.onCanceled;
	}
	this.confirmButton = this.addButton("confirm", this.onConfirmed);
	return view;
};

function DetailsDialog() {
	DialogWindow.call(this);
}

DetailsDialog.prototype = new DialogWindow();

DetailsDialog.prototype.addTitle = function(title) {
	var titleElement = document.createElement("span");
	titleElement.appendChild(document.createTextNode(title));
	this.view.appendChild(titleElement);
};

DetailsDialog.prototype.addDescription = function(description) {
	var descriptionElement = document.createElement("p");
	descriptionElement.appendChild(document.createTextNode(description));
	this.view.appendChild(document.createElement("br"));
	this.view.appendChild(descriptionElement);
};

DetailsDialog.prototype.addDate = function(date) {
	var dateElement = document.createElement("span");
	var dateText = formatDate(date, "DD.MM.YYYY hh:mm");
	dateElement.appendChild(document.createTextNode(dateText));
	this.view.appendChild(document.createElement("br"));
	this.view.appendChild(dateElement);
};

DetailsDialog.prototype.addLocation = function(location) {
	var locationElement = document.createElement("span");
	locationElement.appendChild(document.createTextNode(location));
	this.view.appendChild(document.createElement("br"));
	this.view.appendChild(locationElement);
	return locationElement;
};
```

```
function ItemDetailsDialog(item) {
	DetailsDialog.call(this);
	this.item = item;
	this.profileLink = null;
}

ItemDetailsDialog.prototype = new DetailsDialog();

ItemDetailsDialog.prototype.createView = function() {
	var view = DetailsDialog.prototype.createView.call(this);
	var url = "profile.html?user=" + this.item.user_id;
	var profileLink = this.addLink(this.item.user_name, url);
	var profilePicture = document.createElement("img");
	profilePicture.src = this.item.user_picture;
	profileLink.appendChild(profilePicture);
	view.appendChild(profileLink);
	this.addTitle(this.item.title);
	if(this.item.date_time) {
		this.addDate(this.item.date_time);
	}
	this.addLocation(this.item.location);
	this.addDescription(this.item.description);
	return view;
};

function NotificationDetailsDialog(notification) {
	DialogWindow.call(this);
	this.notification = notification;
}

NotificationDetailsDialog.prototype = new DetailsDialog();

NotificationDetailsDialog.prototype.createView = function() {
	var view = DetailsDialog.prototype.createView.call(this);
	var notificationIcon = document.createElement("span");
	view.appendChild(notificationIcon);
	this.addTitle(this.notification.name);
	this.addDate(this.notification.date);
	this.addLocation(this.notification.location);
	this.addDescription(this.notification.description);
	this.addLink("notifications", "notification.html");
	this.addLink("calendar", "calendar.html");
	return view;
};
```