Automatically Generating User Interfaces for Device Federations

Elmar Braun
Darmstadt University of Technology
Telecooperation Group
Hochschulstr. 10, 64289 Darmstadt, Germany
elmar@tk.informatik.tu-darmstadt.de

Max Mühlhäuser
Darmstadt University of Technology
Telecooperation Group
Hochschulstr. 10, 64289 Darmstadt, Germany
max@informatik.tu-darmstadt.de

Abstract

One of the ideas of ubiquitous computing is that computing resources should be embedded ubiquitously in the environment, making them available to any nearby users. Some researchers have applied this to interaction, and tried to embed an abundance of interactive devices, such as touch screens, in rooms and whole buildings. The opposite concept is that of a single personal mobile device, which users carry at all times and use for all interactions. Because both concepts have different strengths, we explore building interfaces for federations of personal mobile and stationary embedded devices, exploiting the capabilities of both rather than forcing users to choose between either. We have developed an infrastructure that coordinates multiple devices for that purpose: groups of devices work together to render a user interface. As one of the main challenges for such federated user interfaces we have identified their authoring. How should the interface be divided in multiple parts, and can that decision be made by a computer rather than a human designer?

1. Introduction

The concept of ubiquitous computing has been proposed as the natural successor to the age of personal computing. The one-to-one relationship between user and personal computer has the drawback that interaction is bound to a single place. Mobile devices serve only as a "last resort" away from the desktop, and usually are complicated to use in cooperation with personal computers. When many input and output devices are embedded in the environment, as proposed by ubiquitous computing, users should be able to simply associate whatever means of interaction they need.

A first step in this direction was Teleporting [5]: users could walk up to any of the computers distributed throughout a building, use it as if it were their own. Sensors detected the users' presence near a computer, and automatically displayed their personal desktop there. Our initial idea was to extend this still rather desktop-centric concept to arbitrary interaction devices. Instead of a single self-contained device, users associate multiple components, such as screens or speech recognizers, into a *federation* of input and output channels. These components can be both embedded devices from the environment as well as mobile devices carried by the users. This assembly of federations should be context sensitive; for example, voice interaction would not be used in a noisy environment.

What is the benefit of such federations? From the point of view of users of mobile devices, the capabilities of associated devices can be used to work around the limitations inherent to mobile devices. For example, an application that is constrained by small screen size can "overflow" into the additional space of an associated public display that is associated temporarily on demand.

It may seem that environments with generously embedded hardware suffice for convenient interaction, and do not suffer from such limitations. But they too can benefit from associating mobile devices, because there are situations where users cannot rely solely on the infrastructure. Upon closer examination, embedded hardware can be quite limited in many contexts: For example, a large and public display may have no convenient input channel. In such situations associated mobile devices could act as a remote substitute. While their means of input may be far from perfect, they are better than none at all.

The limitation in above example could be solved by replacing the display with a touch screen. But not all limitations are due to device insufficiencies. Input to a touch screen is limited to the small zone from where it can be reached. Mobile devices are usually carried within their user's reach, and could be used to interact with a public screen from anywhere as long as it is in sight. There might also be social restrictions. A large display mounted in a public space might be visible to multiple persons. Any interaction that involves private data cannot be performed on that display. An interface that involves an additional mobile device could be split so that privacy sensitive data remains on the private device, while the public display handles the non-sensitive information.

How should such federated user interfaces, which use multiple devices together, be designed? Authoring applications for desktop computers is easy insofar as the set of input and output channels is always the same. A federation of a mobile device and a public screen has two displays of different sizes, but no keyboard. Other federations may contain even more exotic channels such as speech recognition. And mobile users should be able to (dis-)associate devices on demand, meaning that the federation used to render a user interface may change at runtime. In a federated user interface ideally each of the cooperating devices should specialize on rendering that aspect of the user interface which it is best suited for.

We have developed an infrastructure that allows rendering federated user interfaces on multiple devices of different modalities, and supports changing the set of devices at runtime. This system is described briefly in section 3. Our main contribution is a single authoring scheme that automatically generates federated user interfaces, even for unusual sets of devices, out of a single device independent source. This approach is detailed in section 4. But first we will explore how federated user interfaces relate to other forms of multiple-device use in section 2.

2. Characteristics of Federated Interaction

In the last section we have introduced the concept of federated user interfaces. We refer to a heterogeneous device set, such as the combination of a handheld and a wall mounted display, as a *federation*. We call user interfaces that are adapted to exploit federations *federated user interfaces*. In this section we will discuss what the main traits of federated user interfaces are, and how they differ from other forms of multiple-device use.

The first trait of federated interaction is, as its name implies, that it deals with interactive tasks. This differentiates it from many approaches which combine mobile and stationary devices, but not for interaction. Such concepts are often summarized under the generic term “surrogates”. The concept of surrogates is that limitations of mobile devices are circumvented by offloading tasks exceeding their abilities to stationary devices, called surrogates, in the surrounding environment. Examples are computationally intensive tasks, or tasks requiring more memory than a mobile device can provide. Federated interaction applies the idea of surrogates to interactive instead of non-interactive and invisible tasks.

The second trait of federated user interfaces is that two or more devices are used concurrently, rather than sequentially. Sequential use of devices has also been proposed as a way of supporting mobile users who roam in environments augmented with public devices. Teleporting, which is one example of this, has already been mentioned above [5]. But federated devices deal with using two or more devices at the same time. Of course concurrent use does not rule out sequential use of multiple devices. A user might roam from one federation of devices to another, or between a federation and single devices. The system that we describe in section 3 supports this.

The third trait of federated interfaces is that the interface is distributed to the devices in such a manner that all devices play a role in the interaction. This differentiates federated interfaces from some concepts that focus on remote control. Some researchers have proposed to use mobile devices as a universal remote control that works with any appliance (e.g. [13]). Their goal is to make all functionality controllable from a mobile device, no matter whether the controlled device is present. The controlled device is not used as an interaction channel. Federated interaction seeks to employ remote (on a small scale, such as in the same room but not within touch range) devices as an additional interaction channel. A device within reach of the user is then used for a “remote control” interface that allows interaction with the remote device. But this remote control does not replace the remote device for the purpose of interaction, but instead cooperates with it.

The fourth trait of federated interfaces is that the devices in a federation are heterogeneous. Our reason for combining multiple devices to cooperatively render a user interface is that the weaknesses of one device may be made up for by the strengths of another. For example, when using a cell phone together with a large wall-mounted display, the display offers no privacy and possibly also no convenient mode of input, whereas the cell phone is limited by its small display size. The combination of both can combine the privacy and the input means of the cell phone with the display space of the large display. In this sense federated interfaces are a form of multimodality: the cell phone display and the wall-mounted display have the same physical modality, but apart from that their characteristics are so different that they can be considered two different modalities. Besides physical modality, the distinguishing characteristics between devices include for example mobility (fixed/mobile), ownership (private/public), privacy (private/public), modality (visual/aural), position (near user/out of reach), and more.

3. An Infrastructure for Federated Interfaces

In order to experiment with federated user interfaces, we have built an infrastructure that can use multiple devices concurrently to render a user interface. The rendering is adapted to the set of available devices in two ways. First, not all devices need to render the same interface: different

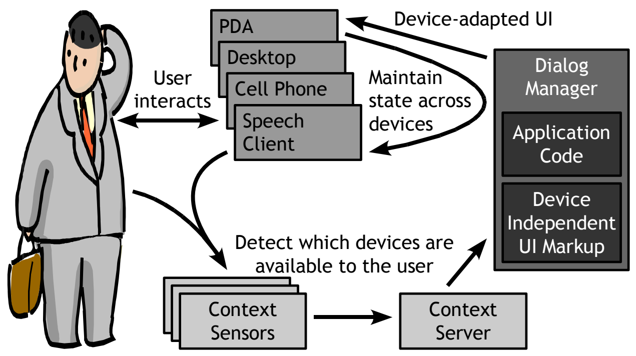


Figure 1. Architecture sketch

devices can render different subsets of the whole interface if appropriate. And not all devices need to render the same part of an interface in the same way: different devices can use different alternate renditions of the same widget. This aspect is examined in more detail in section 4.

3.1. Overview

Our infrastructure for federated devices has three distinct components (see Fig. 1). The first component is a central server, also called dialog manager (DM), that coordinates the interaction with the different devices. The DM also creates multiple adapted versions of the UI for client devices. The second component is comprised of several different clients, which render the user interface and interact with the user. The third component is a context server, which collects and transforms context information from a number of different sensors. Its services are used by the dialog manager.

Overall, our architecture is similar to other architectures that separate the model of an application from its presentation, such as the model-view-controller (MVC) pattern. The main difference is that in our architecture the presentations are not shown on the same device that hosts the model, but distributed to multiple different client devices, while a server hosts the model. Multiple interactive devices render a user interface for the model (or usually different subsets thereof), which visualizes the model and allows to modify it.

This architecture should not be confused with the concept of the X Window system [2]. The updates, which our system sends to the client devices whenever a change in the model occurs, contain the information how the model has changed. The client then uses this information to update its own particular rendition of the model. X instead updates the rendition of the user interface locally on the device where the application runs, and then sends out this update as drawing primitives to its clients. Therefore X cannot support different presentations on different client devices. Even if one connected multiple devices to the same X application, all would display the same window. Our system allows different presentations for different clients. For example, we can use a voice client and a visual presentation at the same time.

Our system should also not be confused with traditional web applications. These do separate the application code, running on an application server, from the presentation, which is shown as a HTML form on a remote client device. One difference is that in our system applications are notified of input events at the same granularity as GUI toolkits. For example, when typing into a text field the application gets a notification for each button pressed. Thus the dialog manager acts like a widget toolkit that is able to place its widgets across several devices. In contrast, web applications only get one event containing the finished form input once the user presses the submit button.

The server in our architecture has a role that is quite different from a web application server. It is the coordinating entity in a form of “virtual browser”, which uses multiple devices concurrently to present different views of the same application. Therefore it is not furnished by a remote application provider, but running on a device near the user, for example one of the client devices, to ensure that feedback to input is shown with low latency. The application runs on the server like an applet in a browser, rather than like a web application.

3.2. Dialog Manager

The dialog manager has four tasks. The first task is to synchronize the different clients when the user interacts with them. It receives input events that change the model from its clients, and forwards the new state of the model to all other clients. For example, if the user selects a checkbox on one device, renditions of the same checkbox on other devices are updated accordingly. Besides the state of the different widgets, the DM also synchronizes the focus. For example, a user can set the focus to a particular widget on one device, and then make speech input to that widget from a different device.

The second responsibility of the DM is to manage users' sessions. The main aspect of that is knowing which devices are available. The DM obtains this information from the context server (see 3.4). The third task of the DM is to host the component that automatically generates user interfaces for device federations out of a single source (see section 4). These two parts of the DM work together to implement context aware roaming. When the context server signalizes that a new device has become available, the DM calls the UI generation component. This call returns user interfaces for the set of devices that a user currently has associated. These are then pushed to all devices including the

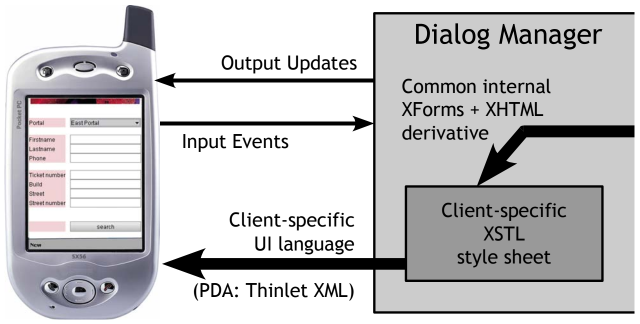


Figure 2. Client operation

newly associated one.

Fourth, the DM hosts the application for which the user interface is rendered. This application is a Java program which is not much different from an application for single-device GUI toolkit with an event loop. All functions described above happen in response to events. The application can subscribe to, modify and generate all types of events that the DM uses. Therefore it is possible to override the automatic behavior of the other components with application-specific code. For example, when a new device is discovered, an application could provide a handcrafted user interface for that device, rather than relying on the output of the UI generator.

3.3. Clients

In order to try out federations with a diverse set of devices, we have developed a number of clients for different modalities. Since we want to be able to support a large variety of different target devices, the effort for writing a new client for a new type of device should be as low as possible. Therefore we attempted to reuse existing user interface renderers as much as possible. The effort for implementing a new client is thus reduced to writing a wrapper around the existing renderer. The wrapper translates between the events that the renderer generates and consumes, and the events that come from and are sent out to the DM.

In order to display a concrete user interface, the device-independent representation of the user interface first has to be transformed into the format that the renderer for a particular device can understand. Therefore a client consists of two parts. Besides the renderer, which runs on the device, a transformation needs to be provided to the server. We use XSLT for these transformations. The transformations are performed on the server (see Fig. 2), because many of the client devices we use have relatively little processing power (e.g. cell phones).

We have developed a number of clients for several different device types. Almost all of them were implemented by wrapping an existing renderer as described above. For large devices, such as desktop computers and wall-mounted displays, we use a web browser. Our first implementation wrapped the Internet Explorer using COM. Because this solution was tied to a single operating system, we developed a second implementation that runs on most web browsers capable of running Java applets (tested with Internet Explorer, Firefox, and Opera). The user interface is pushed to the applet as HTML, which causes the browser to displays it by inserting it into the DOM of the page in which it is invisibly embedded. Input to such a HTML form is intercepted by ECMAScript event handlers, which notify the applet, which in turn forwards the notification to the server.

Since the browser client does not run on PDAs, we developed a separate client for this type of device. It is based on the Thinlet toolkit [1], which works on less sophisticated Java VMs (Java 1.1 or Personal Profile), and uses a proprietary XML language to describe user interfaces. The XSLT stylesheet for the PDA client simply generates Thinlet XML from the abstract user interface fragment. The PDA renderer is only responsible for pushing the user interface to the Thinlet toolkit, and translating between the communication with the DM and the events of the toolkit.

For our cell phone client we could not rely on an existing renderer. However, because of small display size and other limitations of cell phones, a rather simple renderer for user interfaces with few widgets and without sophisticated layout is sufficient. After all, the purpose of this renderer is not to fit the whole user interface on the small cell phone screen, but to display the small part of it, while the rest is rendered by other, larger federated devices. Therefore we wrote a Midlet[1] that directly consumes and displays the device-independent markup used within the DM (a simple subset of XForms and XHTML; see section 4).

In order to include other physical modalities in our experiments, we also developed a client for speech in- and output. The associated XSLT transformation generates a speech interface in the Java speech grammar format (JSGF). The client is a Java program uses the Java speech API, which processes the JSGF document and renders it using an off-the-shelf speech recognizer. In order to be able to provide speech interaction to mobile users, we use a headset that streams audio in- and output to and from a remote desktop computer, which runs the speech recognizer, via WLAN.

3.4. Context Server

The task of the context server is to process and store context information about a user's environment. The DM uses the context server to discover which devices are available to a user. Whenever a new device becomes available, the

[1] A Midlet is an application written for the Java 2 Micro Edition (J2ME) Mobile Information Device Profile (MIDP).

context server sends an event to the DM, which can then automatically push a user interface to that device. The context server, and the sensor infrastructure that is required to make use of it, are not a mandatory component of our system. Associating a new device with a federation can also be done by manually logging on. But a context aware infrastructure can relieve users of this burden, by associating devices automatically when a user's behavior implies that she wants to associate a device.

For example, one of the four different sensor systems that we currently use is IRIS by Aitenbichler [3], which senses head position and gaze direction. This information is used in conjunction with a world model[2] to detect that a user is looking at a device, and automatically associate it. Our other sensor systems include infrared proximity sensing, where small active badges worn by users detect when they are facing similar tags attached to devices [7], and Bluetooth proximity sensing.

The context server is mostly a container for widget-like context filters. Most common context processing tasks are already implemented as generic widgets. Therefore changes to the context processing, such as integrating a new kind of sensor, usually requires only configuring and connecting widgets in a configuration file, rather than programming new widgets.

For example, an IRIS event with head position and gaze direction is first processed by a world model lookup, which transforms this event into an event signifying association between the user and all devices in her view. The next filter stores this event and only forwards it if the association is maintained for more than a few seconds, in order to avoid associating every device a user glances at only briefly. The next widget handles context subscriptions. The DM makes such a subscription to the context server, requesting to be notified whenever a user associates or dissociates a device.

4. Generating Federated Interfaces

One of the main problems for federated user interfaces is their design. Desktop user interfaces are often designed manually. This is possible because most desktop computers are similar with regard to their input and output capabilities. Therefore the designer has to create only a single version of the user interface, and adapt it as much as possible to the average desktop computer.

Such manual design becomes problematic when an interface for more than one target device is required. For example, an interface that is tuned for desktop computers will usually not be usable on a cell phone. But manually designing two interfaces for desktop computers and cell phones means double effort. Manual design therefore becomes very costly for a large number of targeted device types.

[2]The world model is a 3D model of the environment which contains the positions of all stationary devices that are available for association.

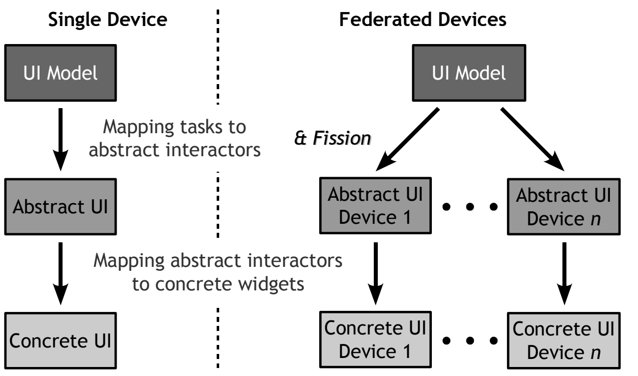


Figure 3. Comparison of development models for single and federated devices

Including device federations as "virtual target devices" aggravates this problem even further. These target devices are more complex than single devices because they usually have a larger set of input and output channels with different characteristics. Also, a large number of supported devices causes an even larger number of possible device combinations. And our system allows users to roam from one federation to another one composed of different devices at the runtime of an application. In order to support such scenarios, a human designer would have to anticipate and make provision for any and all sets of devices that the user might employ to interact.

Because manual authoring is obviously not a viable option for our system, we are investigating the automatic generation of user interfaces for federations from a device-independent source. Automatically generating user interfaces from a single source has already been explored in order to adapt interfaces to multiple different target devices. But these schemes have often focused on adapting downwards: the most powerful supported device is the desktop, and interfaces for other devices are generated by degrading this supposedly most powerful version. A federation can easily be more powerful than a desktop computer. For example, it might have more than one display. The interface generation therefore must be able to adapt upwards to such scenarios. After considering what difference this makes to the interface generation process in 4.1, we will discuss our novel approach to single authoring for federations in 4.2.

4.1. Fission

Figure 3 shows a comparison of the development model for single-device and federated-devices user interfaces. The goal of the development is in both cases a *concrete user in-*

terface. The concrete UI is an interface that can be directly executed without further transformation steps. For example, it could be executable code bound to a concrete widget toolkit, or a concrete markup language for which a renderer exists.

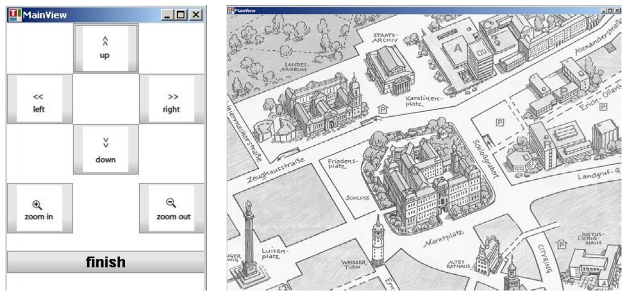
One level above that is the *abstract user interface*, which is not yet bound to a concrete toolkit or markup language. It already specifies which types of interactors are needed, but does not map them to concrete widgets yet. For example, the abstract interface would refer to an abstract "select multiple" interactor where the concrete interface uses a concrete widget such as a "javax.swing.JList" widget or a HTML "<select>" tag.

Above the abstract interface exists another layer, which we refer to as *user interface model*. It contains a definition of the user interface in an even more abstract manner. There are different approaches to such models; frequently used are task models describing the task to be accomplished with the interface, rather than listing individual interactors. Some approaches use a format similar to the abstract interface with some additional annotations; others do not clearly distinguish between the abstract interface and the model. (See [17] for more exhaustive discussions of the topic and related work.)

Of course not all design processes make the layers above the concrete interface explicit to such a high degree. Often, they only exist as concept sketches and (natural language) specification documents. When targeting a single device, this approach is usually sufficient. The basic idea of single authoring (also called single-source, device-independent or multi-device authoring) is specifying these layers in a machine-readable format. This allows to provide tool support or to completely automate the transition from higher levels to lower levels. This transition can then be repeated for different target devices, thereby greatly reducing the effort for creating multiple adapted versions of the interface for different device types.

Designing user interfaces for federated devices adds more complexity to the design process. At some point during the design process, it needs to be decided which parts of the interface are to be displayed on which device. This splitting of the interface is called *fission*. The term fission comes from the area of multimodality, and is the opposite of fusion. Fusion is the process of unifying input from multiple modalities into one logical input event. Fission is the process of dividing one logical output act into several concrete outputs for multiple modalities.

The right part of figure 3 shows our interface design process which includes fission. The mapping from the interface model generates multiple versions of the abstract interface. Each widget or group of widgets can be assigned to a single or multiple devices. The benefit of assigning a widget only to a single device is that it does not take up space on another device, which can then render other widgets. The benefit of assigning a widget to multiple devices concurrently is redundancy, which gives the user more opportunities to perceive and interact with that widget. It is also possible to assign widgets to multiple devices in different fidelities. For example, a large select-one list could be displayed as a drop-down box on a PDA, while an associated large display shows the list as radio buttons.



(a) PDA (b) associated display

Figure 4. Example of fission with two devices

4.2. Using Patterns for Fission and UI Generation

Our first experiments with federated user interfaces explored handcrafted interfaces. The experience from these prototypes was that the question "which widget should be assigned to which device", i.e. the fission, depends on several factors. One factor is of course the device types in a federation. For example, the fission between a large display and a handheld device depends on whether the handheld is a cell phone or a PDA, and whether the display is touch sensitive. Another factor is the application. A large form with many text input fields would be split differently than an application that mostly displays information and requires no input. Other forms of context may play a role as well, as for example whether the user is alone or in public.

The experiences from these designs were recorded in a design pattern format. We refer to them as pre-patterns because the purpose of design patterns is to capture proven solutions, which do not yet exist for federated interfaces. However, the advantages of the pattern format led us to apply it here nonetheless (see [6] for a discussion of the advantages of user interface design patterns). Currently we are validating and refining some of these pre-patterns through user studies.

Figure 4 shows an application to which the simple pattern "Remote Presentation" was applied. The pattern is applicable to user interfaces which consist to a large part of a presentation that does not directly require input, and a smaller part that is used to control that presentation. Examples are applications that display photos, presentation slides, or a map as shown in figure 4. It requires two devices: one device should be a large display; the other can be any device that is convenient for the user to operate, such as a PDA which is carried by and therefore conveniently within reach of the user. The fission strategy of the pattern is simply to assign the presentation to the large display, and put the control part on the other device, which serves as a remote control. This can be seen in figure 4: the PDA renders the scrolling and zooming controls, and the map is displayed by an associated large display.

While the patterns give us tools for the manual design of federated user interfaces, they do not yet solve the problem of automating the fission. The fission rules of a single pattern can be implemented, but since patterns are context-specific, that results in a fission which only applies to that particular context. One option is to let the designer of an interface pick a pattern by hand. This may be appropriate when the pattern choice depends only on the application. But the designer cannot foresee other forms of context, such as which devices are going to be used. We do not believe it to be feasible to require the designer to anticipate all contexts in which an application might be used, and select the correct pattern individually for each situation.

In our approach to fission (see Fig. 5) the pattern choice is not a decision that the designer is required to make, but a step in the automatic interface generation that precedes the fission. The implementation of each pattern does not only consist of an algorithm that assigns widgets to devices, but also of conditions that determine whether a pattern applies to a certain device federation and user interface. The first step of the fission process checks these conditions for all available patterns, and chooses the appropriate ones.

As the base authoring format we use a small subset of XForms [9] and XHTML. In addition, we have a number of custom tags and attributes, which are evaluated by the patterns. This includes: ranking the importance of widgets to decide which to place in the most prominent location; marking widgets as output-only so that they may be placed on displays without convenient input; expressing preferences regarding physical modality, which influence whether a widget may be rendered by voice; expressing relations between widgets, so that widgets that belong together are not torn apart; and specifying privacy preferences for widgets, so that no private information is assigned to a publicly visible (or audible) output channel.

In the next step, each pattern can register its preference for assigning a widget to one or more devices. There are additional auxiliary patterns, which implement guidelines that always should be considered in fission, but are not part of any other pattern. One example for that is the guideline "never assign privacy-sensitive information to a public output channel". Because different patterns may have different preferences, there is a mechanism for ranking their votes. For example, if the map application shown in figure 4 had a private output-only address field, the remote presentation pattern would assign it to the large display, but a higher ranking vote from the auxiliary privacy pattern would make sure that it is assigned to the PDA.

In the next step, the decisions of the different patterns are arbitrated. This step also makes some common sense checks, such as assuring that each widget is rendered by at least one device. Then the abstract interfaces for each device are generated from the output of the fission process. The output format is the same subset of XHTML and XForms as used for the interface model, but stripped of the custom attributes that were only required for the fission.

5. Related Work

The idea of using multiple devices together has first been introduced by Robertson [16]. A number of other projects have also explored using personal mobile devices together with larger stationary devices. Examples are Pebbles [12] and Rekimoto's work on combining handhelds and whiteboards [15]. These projects designed a small number of applications for one fixed set of devices, rather than trying to find a method of automatically generating user interfaces for arbitrary applications on arbitrary device sets.

The concept of a web browser that interacts through multiple devices concurrently has been implemented before by Kleindienst et.al. [10] and Coles et.al. [8]. A similar concept is also outlined in the W3C Multimodal Interaction Framework [11]. However, to our knowledge none of these implementations have provided a solution for the authoring problem. Authoring web interfaces for them usually entails authoring a separate document for each of the involved devices. Changing the employed set of devices during interaction, and therefore the possibility of roaming to other devices, has also not been considered.

The SS/CD project [14] renders multimedia on multiple federated devices, which are automatically chosen from the devices available in the user's environment. The idea of federating devices and selecting those devices based on the user's context is similar to ours. However, the fission of multimedia (such as a movie) is considerably different from the fission of a user interface. Since the modalities of a multimedia presentation are already predetermined, fission is reduced to mapping modalities to the best available output device. For example, the fission of a movie simply means directing the audio output to an audio device, and the video output to a display.

Bandelloni and Paternò have built a system that synchronizes interaction in a federation of exactly two devices [4]. The interface is split up into a control part and a visualization part, which is similar to our remote presentation

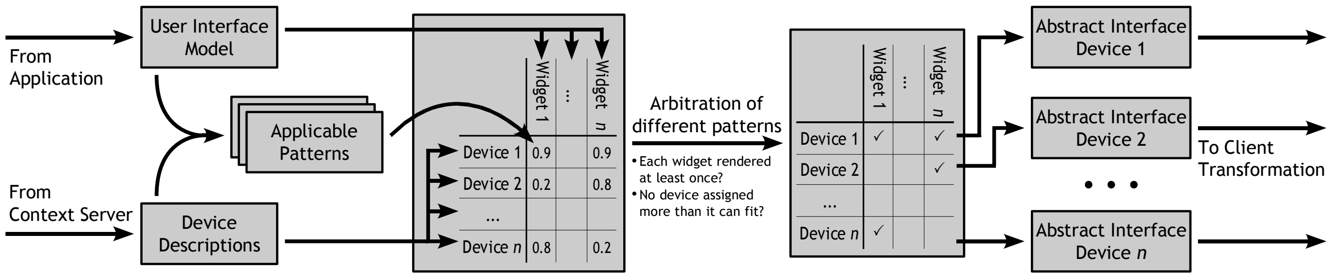


Figure 5. The automatic fission process

pattern. The visualization can migrate to migrate to other devices, while the control cannot roam to other devices. This system uses task models to author user interfaces in a device-independent manner.

6. Summary

In this paper we have addressed the problem of interacting with federations of multiple devices in the "countless appliances pervading the workspace" paradigm of ubiquitous computing. We have presented our architecture which allows seamless roaming between different types of devices and federations thereof. We have also presented a novel scheme for the automatic generation of user interfaces from a device-independent source. This includes automatic fission of a single interface to a federation of multiple devices. Our scheme is based on patterns. Since the set of patterns is not hard coded, existing patterns can be modified and new patterns be added anytime.

References

[1] Thinlet Home Page. `http://thinlet.sourceforge.net`.

[2] X.Org Foundation Home Page. `http://www.x.org`.

[3] E. Aitenbichler and M. Mühlhäuser. An IR Local Positioning System for Smart Items and Devices. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems Workshops (IWSAWC03)*, pages 334–339. IEEE Computer Society, May 2003.

[4] R. Bandelloni and F. Paternò. Flexible Interface Migration. In J. Vanderdonckt, N. J. Nunes, and C. Rich, editors, *IUI '04: Proceedings of the 9th international conference on Intelligent User Interfaces*, pages 148–155. ACM Press, Jan. 2004.

[5] F. Bennett, T. Richardson, and A. Harter. Teleporting – Making Applications Mobile. In *Proceedings of 1994 Workshop on Mobile Computing Systems and Applications*, Dec. 1994.

[6] J. Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons Ltd, Mar. 2001.

[7] E. Braun, G. Austaller, J. Kangasharju, and M. Mühlhäuser. Accessing Web Applications with Multiple Context-Aware Devices. In M. Matera and S. Comai, editors, *Engineering Advanced Web Applications*, pages 353–366. Rinton Press, Dec. 2004.

[8] A. Coles, E. Deliot, T. Melamed, and K. Lansard. A Framework for Coordinated Multi-Modal Browsing with Multiple Clients. In *WWW '03: Proceedings of the Twelfth International Conference on World Wide Web*, pages 718–726. ACM Press, May 2003.

[9] M. Dubinko, L. L. Klotz, Jr., R. Merrick, and T. V. Raman. XForms 1.0, W3C Recommendation 14 October 2003. `http://www.w3.org/TR/2003/REC-xforms-20031014/`, Oct. 2003.

[10] J. Kleindienst, L. Seredi, P. Kapanen, and J. Bergman. Loosely-coupled approach towards multi-modal browsing. *Universal Access in the Information Society*, 2(2):173–188, June 2003.

[11] J. A. Larson, T. V. Raman, and D. Raggett. W3C Multimodal Interaction Framework, W3C Note 06 May 2003. `http://www.w3.org/TR/2003/NOTE-mmi-framework-20030506/`, May 2003.

[12] B. A. Myers. Using Handhelds and PCs Together. *Communications of the ACM*, 44(11):34–41, Nov. 2001.

[13] J. Nichols, B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In *Proceedings of the 15th annual ACM Symposium on User Interface Software and Technology (UIST '02)*, pages 161–170. ACM Press, Oct. 2002.

[14] T.-L. Pham, G. Schneider, and S. Goose. A Situated Computing Framework for Mobile and Ubiquitous Multimedia Access Using Small Screen and Composite Devices. In *Proceedings of the 8th ACM international conference on Multimedia*, pages 323–331. ACM Press, 2000.

[15] J. Rekimoto. A Multiple Device Approach for Supporting Whiteboard-Based Interactions. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI '98)*, pages 344–351. ACM Press, Apr. 1998.

[16] S. Robertson, C. Wharton, C. Ashworth, and M. Franzke. Dual Device User Interface Design: PDAs and Interactive Television. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI '96)*, pages 79–86. ACM Press, Apr. 1996.

[17] A. Seffah and H. Javahery, editors. *Multiple User Interfaces, Cross-Plattform Applications and Context-aware Interfaces*. John Wiley & Sons Ltd, Nov. 2003.