Applying the MVC Pattern to Generated User Interfaces with a Focus on Audio

Dirk Schnelle and Tobias Klug

Telecooperation Group
Darmstadt University of Technology
Hochschulstrasse 10
D-64283 Darmstadt, Germany
{dirk|klug}@tk.informatik.tu-darmstadt.de

Abstract. The mobile user can interact with devices in the environment using either the devices themselves or a device she carries with her. This requires an adaption of the user interface to the used device. Declarative markup languages are considered to be a solution for single authoring user interfaces for different devices and modalities. This is a challenging task, since each device has its own characteristics. We present in this paper a novel architecture to support the creation of user interfaces based on a declarative markup language and a UI independent task model. This architecture is based on the model-view-controller pattern (MVC) to generate user interfaces from declarative markup languages. We introduce a clear border between a modality independent task model and UI design. We also show how the development of smart environments can benefit from the workflow engine underlying our architecture.

1 Introduction

Ubiquitous computing poses new challenges to human-computer interfaces. One of them is the use of input devices that are available in the environment or which the user carries with her. As a consequence it is necessary to adapt the interaction with the user to the current device. This implies also the use of different communication channels, or *modalities*, which are used to interact with the user [3]. The development of multimodal applications is complex and time-consuming, since each modality has unique characteristics [12]. Research and modern development approaches try to solve this by means of declarative markup languages, mostly XML based.

They promise that the same model can be reused for different modalities and output devices [20]. The reusability concerns mainly the so called *business logic* of the application. We understand business logic as a specification of the concept: Who (roles) does what (task) when (process) and how (environment). The main requirements in business logic are reusability and the independence of the user interface. But the community of authors and users of generated interfaces have already discovered the limits of this approach. “Using the same user interface for all devices means that the thinnest device will set the limits

for the user interface, and unless the user interface is extremely simple, some device categories necessarily will be excluded" [15]. Gerd Herzog et al. [8] come to the same conclusion that "it is impossible to exclude all kinds of meaningless data from the language and the design of an interface specification will always be a sort of compromise" [8].

One of the modalities for which such a compromise has to be found is audio. Its invisible and transient nature poses many challenges to the interface designer [19]. Due to the fact, that multimodal applications are often considered to have both, graphical and audio in- and output capabilities, it is astonishing, that audio only applications seem to be considered with lower priority.

Lingam [10] first seems to ignore this fact by stating that the "[...] day the limitations of voice recognition and Natural Language processors are overcome [...]" [10], but he also argues that not all limitations can be overcome. He sees a solution in a complementary use of all available modalities. This is also the common tenor, that voice can only be used as a complementary modality, but not on it's own. Because humans are more visually oriented than aural, there is much more research being done with a focus on graphical rendering, i.e. layout of forms, than on audio based interfaces.

Shneiderman names the example of a stock market where a survey result showed, that although trading is done by voice, the visual approach is 10 times more attractive to users [19]. The limits of audio are not well understood and are therefore replaced by the approach to use audio as a complementary modality. However, under some circumstances audio is a first class medium. Especially for visually impaired people or for workers who do not have their hands and eyes free to interact with the computer.

In this paper, we introduce a novel architecture to support the creation of user interfaces based on a declarative markup language and a UI independent task model. This architecture satisfies all requirements in a reusable business logic and combines it with the advantages of generated user interfaces.

The architecture applies the MVC pattern to generated user interfaces using existing standards i.e. workflow engines and XHTML. Our main goal is to remove all implementation details from the model by separating the application into a modality or implementation independent part executed by the workflow engine and a modality dependent part executed by a renderer for each supported modality. Using a workflow engine as the central building block allows us to easily integrate backend systems, user interfaces and other services like sensors and actuators to the environment via a standardized API.

The rest of this paper is structured as follows. In the motivation section we name shortcomings of existing solutions for UI generation. The next chapter takes a closer look at existing comparable solutions. Then we give an overview about our architecture and the way from the task model to the UI in section 4. Two use case scenarios for voice based user interfaces and graphical user interfaces are given in section 5 to demonstrate how our approach can be applied to different modalities. Finally we conclude this paper with a short summary and an outlook to further enhancements.

2 Motivation

Declarative markup languages use the MVC pattern [6] to decouple model and business logic from representation and user interaction. Luyten [11] concretizes this with his model based approach. He distinguishes between *task model* $\mathcal{M}_\mathcal{T}$, *presentation model* $\mathcal{M}_\mathcal{P}$ and *dialog model* $\mathcal{M}_\mathcal{D}$. These can be directly mapped to *model*, *view* and *controller* of the MVC pattern [6]. From the MVC's perspective, controller and view are responsible for the presentation. The model can be reused for alternative representations. Some UI generators for declarative languages also reuse the controller. But since the controller is tightly coupled with the view, it is debatable if this approach can be successful for different modalities. Others try to generate the dialog model out of the task model [11]. Luyten in fact tries to generate a mapping

$$\mathcal{M}_\mathcal{T} \rightarrow \mathcal{M}_\mathcal{D} \tag{1}$$

From the MVC point of view the controller serves as a mediator between task model and presentation model but we doubt that it can be generated from the task model. Since the model contains no clues about the interface this attempt will result in basic interfaces that need further editing. An example for that is the form generator of Microsoft Access, that creates a basic form UI from a table. However, the MVC approach seems to be promising, but we still need different dialog models for different modalities. These approaches, like the ones of Paterno and Luyten, are widespread in academic research but not in industry. One reason, besides the known limitations of generated interfaces, is that they primarily focus on the interaction with the user and only some even consider integration into back-end systems, which is a requirement for the business case.

Consider a brokerage application, where the system *displays* a huge list of stock market values. This is no problem for visually oriented modalities, even on small displays. The one-dimensional and transient nature of audio makes the delivery of large amounts of data nearly impossible. Audio requires different interaction strategies. Another problem is caused by the accuracy of recognizers. Since speech is not detected with an accuracy of 100%, additional steps are necessary to confirm the entered data. A big advantage of speech, to provide many information items at once remains unexploited by existing UI generators. This feature can be used in mixed initiative dialogs with the requirement:

Say what you want at any time you want to.

Mixed initiative dialogs allow for overloading, which means, that the user is able to provide information that is not in the focus of the current dialog step. This is not possible in graphical UIs or with task based generators collecting this information from the user sequentially.

It is obvious, that the reuse of the dialog model is not suitable to satisfy the needs of different modalities, which is probably also a reason, why this model is the one least explored [16]. Olsen's statement is still true as Luyten points out in [11].

This becomes clearer in the following example. Consider a shopping task, where the customer first has to select the items to buy and then proceeds to entering the billing address. Figure 1 shows, how the task *Shopping* is decomposed into the sub-tasks *Select an item* and *Purchase the items*. This is exactly the way as it is proposed by Luyten et al. [11]. According to Luyten, a task t can be recursively decomposed into a set of sub-tasks:

$$t \xrightarrow{d} \{t_1, \ldots, t_n\}_{n \geq 2} \tag{2}$$

We concentrate on the purchasing task, which is further decomposed into the sub-tasks *Enter address information* and *Enter credit card information*.

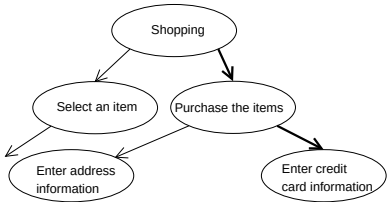

Fig. 1. Shopping Task

The designer of a graphical interface would stop modelling at this stage, since these are tasks, which can be handled by a single view in the user interface. Note that it is also possible to expand these tasks, but it is not necessary. An example implementation of the address entry from Amazon is shown in Figure 2.

The same level of expansion would not work in audio, since each input field of the address entry is a complex operation in audio and has to be treated separately.

Apart from the problem, that free form voice entry of names is still an unsolved problem, several other challenges inherent to the medium have to be faced. Since speech cannot be recognized with an accuracy of 100%, users need feedback about the entered data. Another problem is, that speech is invisible, so users might forget what they entered few moments ago. As a consequence, the task *Enter address information* must be expanded for voice as shown in Figure 3. Stopping decomposition is no longer an option.

Another problem deals with the selection from lists, i.e. selecting the country. Since speech is one-dimensional, it is unacceptable for users to listen to a list with dozens of items. This requires special interaction techniques, like scripted interaction [18].

This means, that a task decomposition d may have media dependencies M, where M is a set of targeted media m_i.

Equation (2) has to be extended to:

$$t_M \xrightarrow{d} \{t_{1,M'}, \ldots, t_{n,M'}\}_{n \geq 2} \tag{3}$$

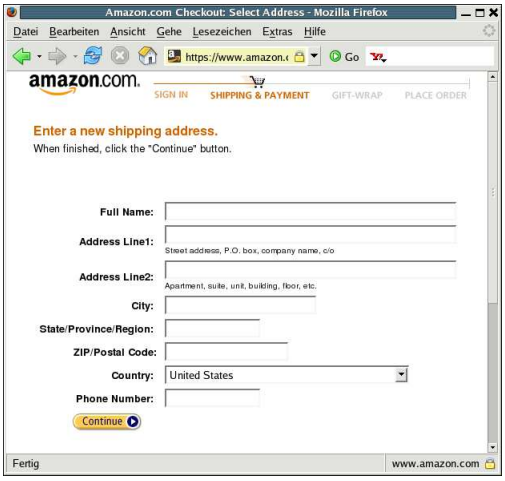

Fig. 2. Amazon: enter address information

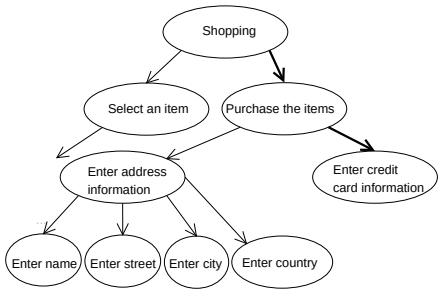

Fig. 3. Expansion of Enter address information for voice

where M' is a set of supported media of the decomposition with

$$M' \subseteq M \tag{4}$$

In this paper, we introduce an approach that splits a set of tasks into a task model and a dialog model. The task model is highly independent of the modality whereas the dialog model captures modality specific aspects. This approach is used to create higher quality UIs and satisfies all requirements in a reusable business logic. This means to stop task decomposition at a point where $M = \hat{M}$ with $\hat{M}$ including all known media, which means independence of media.

All tasks t_M with $M \neq \hat{M}$ must not be part of $\mathcal{M}_\mathcal{T}$ but part of $\mathcal{M}_\mathcal{D}$. More mathematically

$$\mathcal{M}_\mathcal{T} = \{t_{M'} | \forall t_{M'} \in T : M' = \hat{M}\} \tag{5}$$

3 Related Work

Chugh et al. have reviewed the influence and advantages of voice technologies on existing web and enterprise applications [4]. They introduce an architecture built upon a centralized application server providing services through a CORBA API. One of the supported UIs is a phone based access via a VoiceXML interpreter.

This architecture has the business logic in CORBA services. These services are more or less decoupled. The order in which they are used depends on the logic implemented in the client. This means, that parts of the business logic are shifted from the model to the controller. What is missing, is a structured way to separate real business logic from presentation logic. Our approach uses a workflow engine for sequencing service calls.

In [23] Vantroys et al. describe an architecture of a learning platform with a centralized workflow engine. The targeted platforms vary from desktop PCs to mobile phones and their focus is to transform XML formatted documents stored in the learning database into a suitable format for the current device. This is achieved by XSL transformations. Since they use their own proprietary format for the documents this approach is strongly limited. They consider neither a general approach nor the way how users can access the documents. The main disadvantage is that they do not consider user input at all. In addition the approach takes no respect to special limitations of the used device. They stop at transforming into a format that is supported by the target device.

In [13] Mori et al describe an approach called TERESA, where the *Task Model* is being used as a common basis for platform dependent *System Task Models*. In contrast to our approach their task model contains a mix of modality dependent and modality independent tasks. By filtering they then generate a *System Task Model* for each modality. Because of that, the main *Task Model* needs to be a compromise between different modalities. We believe that this approach does not support the design of efficient modality specific dialog models. Mori's Task Model allows the designer to specify targeted platforms and even allows to store dependencies among these platform descriptions. This becomes evident in their

discussion of a demo implementation. The VoiceXML enabled system plays a *grouping sound*, which makes only sense in visual environments. This sound does not provide any information to the user of audio-only interfaces, but is transferred directly from the visual interface without questioning its sense. This is not compliant with our understanding of a model. We store only abstract task descriptions without any relation to possible implementations.

One of the main problems when dealing with (semi-)automatically generated user interfaces is the mapping problem defined by Puerta and Eisenstein [17]. The mapping problem is characterized as the problem of "linking the abstract and concrete elements in an interface model". However, our approach does not deal with the mapping problem as the creation of modality dependent dialog models is left to the designer.

4 Architecture

4.1 General

Today, the established term of *workflow systems* comes back as *business process modelling* systems. The **W**ork**f**low **M**anagement **C**oalition, WfMC [21], has defined *workflow* as the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another, according to a set of procedural rules. Workflows are established in industry and provide methods to describe and access process definitions. They help to coordinate multiple applications with the defined activities. The sequence in which these activities are performed is controlled by a workflow engine. It also defines who (roles) does what (task), when (process) and how (environment). An advantage of process descriptions for workflows is that they do not consider UI components by definition. We have an independence of media. Thus they fulfill equation (5).

User interfaces for workflow engines have been considered to be graphically oriented desktop PC applications only. The MOHWAS [14] project extended this approach to mobile users using heterogeneous devices. Moreover it tried to integrate contextual information from the environment into workflows. However, research in this project concentrated on graphical devices. Our approach tries to fill the gap between the world of workflow engines and the world of voice based UIs.

4.2 Workflow Process Description

Our basic architecture is shown in figure 4.

The task model is stored as *Workflow Data* and is accessed by the *Workflow Engine*. The *Controller* reads the current activity from the workflow engine and asks the *Renderer* for the next UI. Currently the output of the Renderer is limited to markup languages. But this is only a minor issue, since markups for Swing based Java user interfaces and others exist [1] and are well explored.

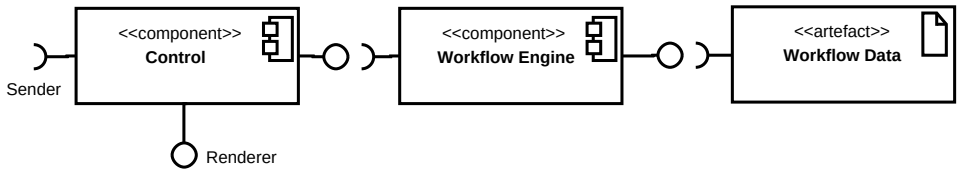

Fig. 4. Core architecture

The workflow engine uses the XPDL format [22] to describe processes and their activities. We will explain the architecture using a shopping process example shown in Figure 5.

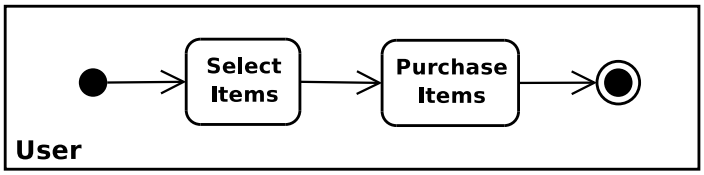

Fig. 5. Shopping task

To author processes in XPDL, we use JaWE [5]. The following listing in XPDL format shows the activity *Add to cart* from the *Select Items* block activity shown in Figure 6 to select the items to buy.

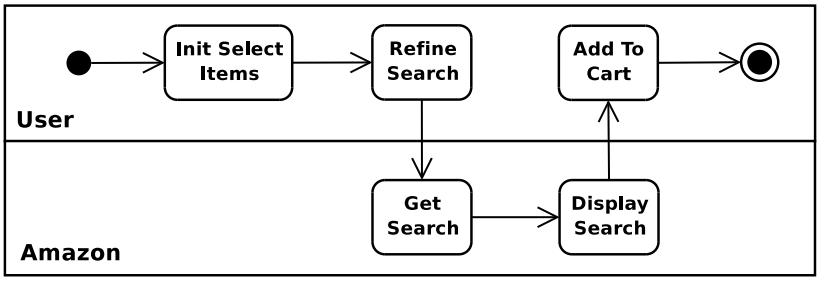

Fig. 6. Shopping task: Select Items

```
<Activity Id="question" Name="Add to cart">
    <Implementation>
```

```
        <Tool Id="SendData" Type="application">
            <ActualParameters>
                <ActualParameter>user</ActualParameter>
                <ActualParameter>booklist</ActualParameter>
                <ActualParameter>cart</ActualParameter>
            </ActualParameters>
        </Tool>
        <Tool Id="ReceiveData" Type="application">
            <ActualParameters>
                <ActualParameter>user</ActualParameter>
                <ActualParameter>booklist</ActualParameter>
                <ActualParameter>cart</ActualParameter>
            </ActualParameters>
        </Tool>
    </Implementation>
    ...
</Activity>
```

Parameters of the task are *user* (the user's identification), *booklist* (a list of books, which has been obtained from a database) and *cart* (the items, which the user selected). The *Control* component is attached as a so called *ToolAgent* and exchanges the data with the UI.

In the example, we concentrate on voice based user interfaces. We use VoiceXML to handle speech input and output. Other modalities can be supported by other markup languages, see section 5.1. Figure 7 shows the architecture of the VoiceXML renderer.

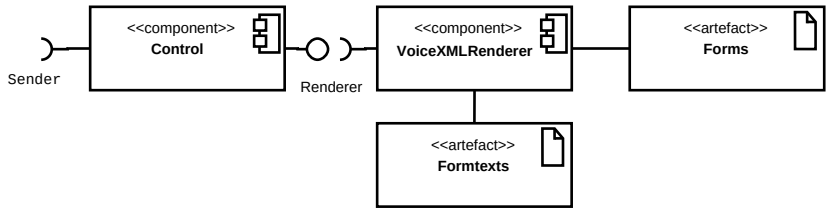

Fig. 7. VoiceXML Renderer based on XForms

The *VoiceXMLRenderer* implements the *Renderer* interface of the *Control* component. The renderer transforms XForms [24] forms, stored in the correspondent artefact into the targeted markup language, i.e. VoiceXML.

Each activity of the workflow engine can have multiple forms. This enables a modality dependent decomposition into sub-tasks as shown in figure 3. In fact the *Control* component is a state machine, that handles all interactions belonging to a single activity. The process designer can concentrate on her task to establish

a workflow and has little stimulation to over design the workflow, since she need not care about the UI.

4.3 Modelling of Data

XForms is based on an `<xfm:model>` element. The following listing shows the model for the book list:

```
<xfm:model>
    <xfm:instance id="booklist">
    <data>
        <title>
            Harry Potter and the halfblood prince
        </title>
        <author>J. K. Rowling</author>
        <review>
            The long-awaited, eagerly anticipated, arguably
            over-hyped Harry Potter...
        </review>
        <price>17.99</price>
        <quantity/>
    </data>
    </xfm:instance>
    ...
</xfm:model>
```

We use the workflow engine to retrieve the data. In our case the data is obtained from a database in the *Get Data* activity and stored in the *booklist* parameter of the shopping process.

4.4 Definition of Forms

Next we have to define that the title and price should be read to the user, before she is prompted for the quantity. This is shown in the next listing.

```
<xfm:output id="title" ref="instance('cart')/data/title"/>
<xfm:output id="author" ref="instance('cart')/data/author"/>
<xfm:input id="quantity"
    ref="instance('cart')/data/quantity">
```

Besides simple input and output for the default data types known from XPDL, we also implemented support for selections from lists and complex data types like structures.

The `ref` attributes indicate, that the text resides outside this document. This can be used i.e. for internationalization. It is defined in its own document. This makes it possible to write texts independently of the XForms data model and the workflow activity. The following listing shows the text for the shopping task.

```
<Item isa="text">Do you want to buy</Item>
<Item isa="output" id="title"/>
<Item isa="text">from</Item>
<Item isa="output id="author"/>
<Item isa="text">for</Item>
<Item isa="output id="price"/>
```

In addition, a form can have internal variables which are used to control the selection of the next form or to transfer data from one form to another.

4.5 The Form State Machine

Next we have to define how to process the data. An action to perform such a processing step is defined by the `<xfm:action>` tag. The XForms specification defines elements like `<xfm:trigger>` or `<xfm:submit>` to activate certain actions like `<xfm:setvalue>` or `<xfm:insert>`. Some of these action elements do not have a meaningful implementation in voice based UIs. One of them is `<xfm:setfocus>`. In our prototype, the voice renderer simply ignores these tags. Again it is noteworthy that the weakest medium limits the expressions of a UI markup language. Current implementations ignore this or are just able to produce UIs with limited capabilities [8].

An action to perform such a processing step is defined by the `<xfm:action>` tag. The following listing shows how this action is defined in our example to add a book from the booklist to the shopping cart.

```
<xfm:action>
    <xfm:insert at="1" ref="instance('cart')/cart/order"
        position="before"/>
    <xfm:setvalue ref="instance('cart')/cart.../@amount"
        value="'1'"/>
    <xfm:setvalue ref="instance('cart')/cart.../@price"
        value="instance('booklist')/books/..[...]/@price"/>
</xfm:action>
```

We look upon the forms as a description of input and output elements, linked with the actions. Each activity defined in the workflow engine can have multiple assigned forms. The logic to *display* a certain form is then comparable to a state machine. This is where we enhance XForms with the `<Transition>` tag. After submission of a form, the state machine decides, whether there is another form to display or if the activity is complete. In the latter case, control is returned to the workflow engine. A transition can be defined like in the following example:

```
<Transitions>
    <next to="bookinfo">
        <test var="todo" op="eq" value="n"/>
    </next>
    <next to="booklist_again"/>
</Transitions>
```

Checking for the next next transition is done by the `eq` attribute of the `<next>` tag. Currently we are able to check data for equality in combination with logical operators `<and>` and `<or>`, which can surround the `<next>` tags. An empty `<next>` tag is the default value, where control is returned to the workflow engine.

Some media may require certain renderer parameters, like fonts and colors for XHTML or voice for VoiceXML. These parameters depend on the renderer. They can be defined in a `<RendererParam>` node. They are delivered to the renderer without further evaluation.

The benefit from using a state machine is that the UI tasks are separated from the business process. We gain a separation of concerns. The business process is defined in the workflow engine without any information of the display medium. UIs are defined in an independent repository with all the specific parameters that are required for rendering.

4.6 VoiceXML Frontend

The Renderer can be defined via a plug-in architecture. As an example implementation, we show a VoiceXML renderer based on XForms (Figure 8).

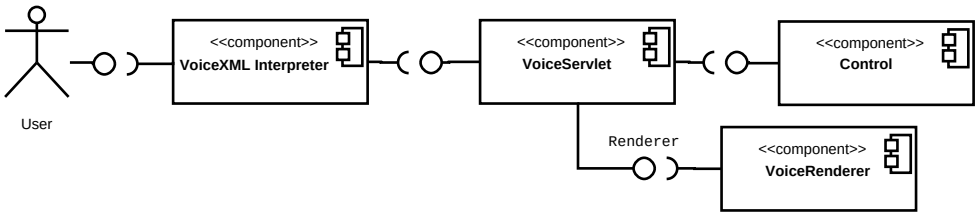

Fig. 8. VoiceXML Frontend

The VoiceXML renderer is implemented using an XSLT transformation. We use the Saxon [9] implementation, since this implementation satisfies all our needs and is fast enough.

The user communicates with the VoiceXML interpreter over the phone. The VoiceXML documents are retrieved from the VoiceServlet, stored on a Tomcat servlet container. The VoiceServlet communicates with the Controller, using MUNDO [2, 7], a communication middle-ware developed at the telecooperation group of the Technical University of Darmstadt.

5 Scenarios of Use

To illustrate how this architecture can be applied to ease the development of ubiquitous computing systems, we will discuss how it could be used in two exemplary scenarios.

The first scenario is a ticket machine which overcomes the bottleneck of its single graphical user interface by allowing different modes of access depending on the resources available to the user.

The second scenario highlights the usefulness of the workflow capabilities introduced by the architecture. This example demonstrates how user input and information gathered from sensors in the environment can be easily integrated into a single application.

5.1 Ticket Machine Scenario

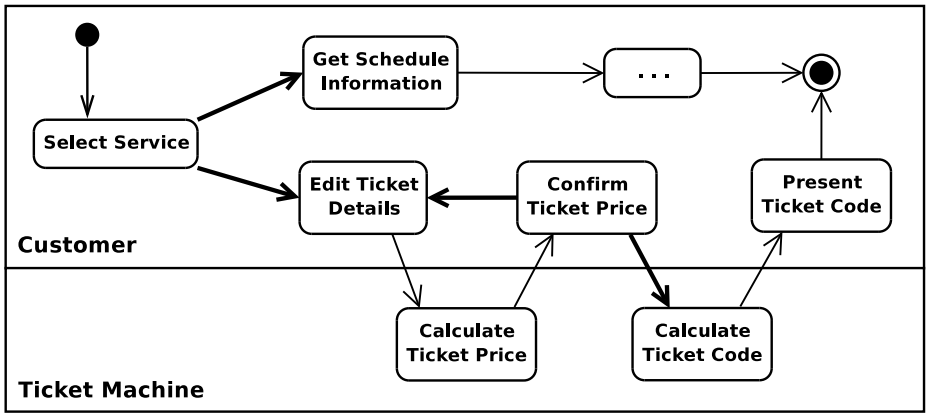

Fig. 9. Ticket Machine workflow example (bold arrows indicate conditional transitions)

We all know the situation. We have been rushing to the station to catch a specific train. As we arrive at the platform, the train is already pulling into the station. Unfortunately we still need to buy a ticket and the queue in front of the ticket machine kills any thought of taking that train.

The bottleneck in this situation is the ticket machine. More specifically it is the fact that there is only a single, sequential working user interface to the ticket machine which has to be shared by all users. Imagine a ticket machine, that is able to extend its user interface to external devices like PDAs, mobile phones or bluetooth headsets. A user that wants to use the ticket machine can connect to it wireless with her device. The machine determines the capabilities of the mobile device and renders an appropriate user interface.

The first activity asks the customer which service she wants to use. Depending on her selection, the *get schedule information* or *edit ticket details* tasks are executed. For simplicity the remainder of the *get schedule information* workflow has been omitted. *Edit ticket details* presents default ticket details to the customer and gives her the opportunity to change them. When she finishes this

activity, the values are transferred to the *calculate ticket price* activity which calculates the final fare for the ticket. Next, the calculated amount is presented to the customer who can decide to make further adjustments or confirm the purchase. If she confirms, the ticket machine computes a ticket code which is again presented to the customer. Once this is done, the workflow finishes.

Although different interaction techniques (e.g. VoiceXML, XHTML) are supported, they all share the same basic workflow to determine the overall structure of the application and how information is exchanged between different activities. This scenario demonstrates key strengths of the architecture provided. By decoupling the business logic from the implementation of the user interface, the implementation of different modalities is greatly simplified while still providing enough flexibility for the UI designer to build the interface most appropriate for a specific interaction technique. Also by having the same workflow beneath all variants, the user does not have to relearn the logic behind the application when switching devices.

Voice-Only Interface The voice-only interface can be used when no visual interface is available. Low-end mobile phones or wireless headsets would be examples of devices using this mode of access. This is also true when visual user interfaces are not an option as for visually impaired people.

Sample Dialog:

System: Which Service can I provide? Say 1 for buying tickets and 2 for schedule information.
User: 1

System: Do you want an adult ticket for zone 1? Say 1 to change the ticket type, 2 for another zone and yes to confirm.
User: 1
System: Name the number of the destination zone.
User: 1
System: Say 1 for adult, 2 for student and 3 for child.
User: 2
System: Do you want a student ticket for zone 1? Say 1 to change the ticket type, 2 for another zone and yes to confirm.
User: yes

System: The student ticket for zone 1 costs €1.20, buy that ticket?
User: yes

System: Your ticket code is ABZ723S. Shall I repeat the code?
User: yes
System: Your ticket code is ABZ723S. Shall I repeat the code?
User: no
System: Good bye and thank you for using our service.

Notice that the architecture is flexible enough to accommodate UI patterns specific to the voice based nature of the interface. I.e. repeating important information is one of these aspects that is not necessary with visual interfaces, but which is required in audio based interfaces due to it's transient and invisible nature.

The task model could also be used for a hand crafted mixed initiative dialog.

System: What can I do for you?
User: I need a ticket for zone 1
System: Do you want to buy an adult, a student or a child ticket for zone 1?
User: student

System: Do you want to buy a student ticket for zone 1 for €1.20?
User: Yes

System: Your ticket code is ABZ723S. Shall I repeat the code?
User: No.
System: Good bye and thank you for using our service.

This dialog is only hand crafted, but demos the flexibility of our approach. It is even possible to generate user interfaces like that with another type of renderer

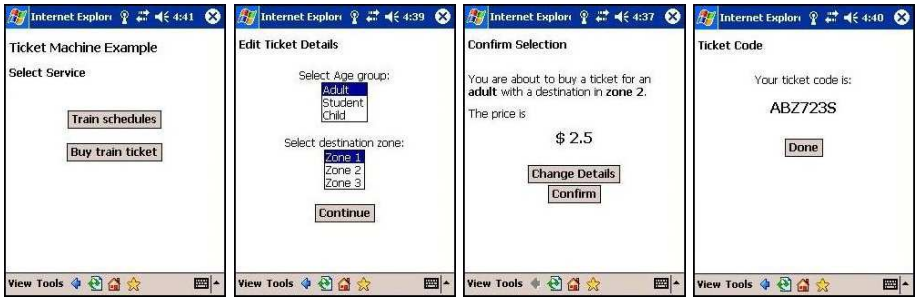

Fig. 10. Ticket Machine Example XHTML Interface

XHTML Interface When a PDA or smartphone is available, the architecture uses a different renderer with its own repository of UI forms to deliver the service to the customer. Figure 10 shows sample screenshots from an XHTML renderer.

The forms for this renderer are been optimized for PDA and smartphone interfaces to improve the user experience. At the same time, the overall business process and associated links into the backend system are reused. Therefore the user interface designer can concentrate on the user experience and is not concerned with backend integration.

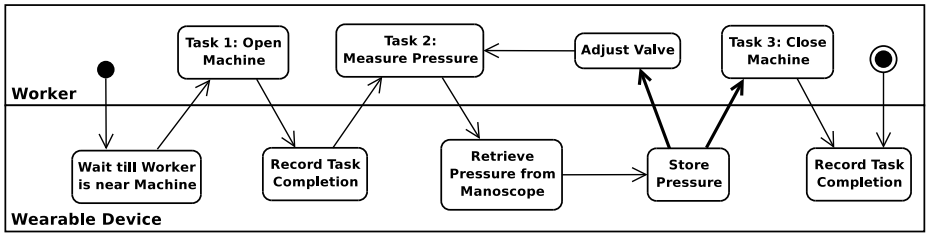

Fig. 11. Maintenance example workflow (bold arrows indicate conditional transitions)

5.2 Maintenance Scenario

This example implements a maintenance task for a complex machine, that involves opening the machine, using a manoscope to measure a pressure, adjusting a valve to regulate that pressure, and closing the machine again. The worker uses a wearable device to get instructions on the task and record its performance. The manoscope is connected to this device and can transfer measurements directly into the application. Task execution starts, when the wearable device detects that the worker is close to the machine.

Again the central part of this application is a workflow definition, that connects and orchestrates the different components of the software. These components are the user interface to give instructions to the user, a component to detect the user's proximity to the machine and a component that communicates with the pressure tool.

The workflow starts with an action for the proximity component. This task is finished as soon as the user is close to the machine to be maintained. Next the worker is instructed to open the machine and confirm this step. The system then documents this step of the maintenance task for future reference. Now the worker is instructed to measure the pressure using the pressure tool. Once the measurement is taken and confirmed by the user, the system retrieves the measure from the tool and stores it for documentation. Depending on the pressure measured, the user is instructed to adjust a valve and measure the pressure again. This is repeated until the pressure is within a certain range, at which point the user is asked to close the machine again, which concludes this maintenance task.

If the wearable system was able to detect the open and close machine tasks by analysing further sensors in the environment, these steps could also be automated.

This scenario shows how smart devices and context information can be integrated into applications. By using the workflow as a basic structural model of the application, the code for smart devices, context detection and user interface can be separated.

6 Conclusion

In ambient environments, multiple devices are used to interact with the environment. Each device has it's own characteristics, depending on the modalities used. Declarative markup languages are considered to be a solution for single authoring user interfaces targeting different devices and modalities. In this paper we introduced a novel architecture to combine model based UI generation from declarative markup languages with a modality independent task model.

We showed that current approaches of task models are not modality independent and pointed out, that this is the main reason, why these approaches for UI generation are not yet widely adopted in industrial settings. Our approach is a generalization of the MVC pattern to model based architectures as proposed in [11].

We identified modality dependent components of the task model and moved those to the dialog model. As a positive consequence, the business logic, which is stored in the remaining task model, is fully reusable for different modalities. This makes it also possible to integrate other systems or trigger events.

We implemented our architecture and showed, that it is possible to generate higher quality user interfaces which take respect to the current modality. The current implementation focuses on voice interfaces. Other markup languages like XHTML are currently being implemented (see section 5.1).

In the near future we will explore the quality of these generated interfaces in user studies. Furthermore the current approach is restricted to markup languages. We need to find a way, how to overcome this limitation and find a way to generate more flexible UIs i.e. mixed initiative dialogs.

Another problem to solve is the need to figure out which device to use to interact with the user by means of the ambient environment.

Acknowledgments

Thanks a lot to Dr. Matthias Lipp from danet GmbH `http://www.danet.com` for his patient support and troubleshooting our problems with the workflow engine WfMOpen `http://wfmopen.sourceforge.net`.

References

1. Abstract User Interface Markup Language Toolkit. `http://www.alphaworks.ibm.com/tech/auiml`, March 2006.
2. Erwin Aitenbichler, Jussi Kangasharju, and Max Mühlhäuser. Experiences with MundoCore. In *Third IEEE Conference on Pervasive Computing and Communications (PerCom'05) Workshops*, pages 168–172. IEEE Computer Society, March 2005.
3. Jürgen Baier. Beschreibung von Benutzerschnittstellen mit XML. Master's thesis, Fachhochschule Karlsruhe, Fachbereich Informatik, 2001.

4. J. Chugh and V. Jagannathan. Voice-Enabling Enterprise Applications. In *WET-ICE '02: Proceedings of the 11th IEEE International Workshops on Enabling Technologies*, pages 188–189, Washington, DC, USA, 2002. IEEE Computer Society.
5. Enhydra. Open Source Java XPDL editor. http://www.enhydra.org/workflow/jawe/. accessed on 05/08/2006.
6. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1992.
7. Andreas Hartl, Erwin Aitenbichler, Gerhard Austaller, Andreas Heinemann, Tobias Limberger, Elmar Braun, and Max Mühlhäuser. Engineering Multimedia-Aware Personalized Ubiquitous Services. In *IEEE Fourth International Symposium on Multimedia Software Engineering (MSE'02)*, pages 344–351, December 2002.
8. Gerd Herzog, Heinz Kirchmann, Stefan Merten, Atlassane Ndiaye, and Peter Poller. Multiplatform testbed: An integration platform for multimodal dialog systems. In Hamish Cunningham and Jon Patrick, editors, *HLT-NAACL 2003 Workshop: Software Engineering and Architecture of Language Technology Systems (SEALTS)*, pages 75–82, Edmonton, Alberta, Canada, May 2003. Association for Computational Linguistics.
9. Michael Kay. `http://saxon.sourceforge.net/`. accessed on 05/12/2006.
10. Sumanth Lingam. UIML for Voice Interfaces. In *UIML Europe 2001 Conference*, March 2001.
11. Kris Luyten. *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development*. PhD thesis, transnational University Limburg: School of Information Technology, 2004.
12. J. Terry Mayes. *Multi-media interfaces and learning*, chapter The 'M-word': multimedia interfaces and their role in interactive learning. Springer-Verlag, Heidelberg, 1992.
13. Giuli Mori, Fabio Paternò, and Carmen Santoro. Design and Development of Multidevice User Interfaces through Multiple Logical descriptions. *IEEE Transactions on Software Engineering*, 30(8):507–520, August 2004.
14. Mowahs project. `http://www.mowahs.com/`, 2004.
15. Stina Nylander. The ubiquitous interactor - mobile services with multiple user interfaces. Master's thesis, Uppsala: Department of Information Technology, Uppsala University, November 2003.
16. Dan Olsen. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufman Publishers Inc., 1992.
17. A. Puerta and J. Eisenstein. Towards a general computational framework for model-based interface development systems. *Proceedings of the 4th international conference on Intelligent user interfaces*, pages 171–178, 1998.
18. Dirk Schnelle, Fernando Lyardet, and Tao Wei. Audio Navigation Patterns. In *Proceeding of EurPLoP 2005*, 2005.
19. Ben Shneiderman. The Limits of Speech Recognition. *Communications of the ACM*, 43(9), 2000.
20. Jonathan E. Shuster. Introduction to the User Interface Markup Language. *CrossTalk*, pages 15–19, January 2005.
21. The Workflow Management Coalition. Woorkflow Management Coalition - Terminology & Glossary. Technical Report Technical Report WMFC-TC-1011, The Workflow Management Coalition (WfMC), February 1999.
22. The Workflow Management Coalition. Workflow Process Definition Interface – XML Process Definition Language (XPDL). Technical Report WFMC-TC-1025, The Workflow Management Coalition (WfMC), October 2005.

23. Thomas Vantroys and José Rouillard. Workflow and mobile devices in open distance learning. In *IEEE International Conference on Advanced Learning Technologies (ICALT 2002*, 2002.
24. W3C. `http://www.w3.org/TR/xforms11/`. accessed on 05/09/2006.