Inf Syst Front
DOI 10.1007/s10796-007-9028-5

Specifying and controlling multi-channel web interfaces for enterprise applications

Matthias Book · Volker Gruhn

Abstract When building enterprise applications that need to be accessed through a variety of client devices, developers usually strive to implement most of the business logic device-independently while using a web browser to display the user interface. However, when those web-based front-ends shall be rendered on different devices, their differing I/O capabilities may require device-specific interaction patterns that still need to be specified and implemented efficiently. We present an approach for specifying the dialog flows in multi-channel web interfaces with very low redundancy and introduce a framework that controls web interfaces' device-specific dialog flows according to those specifications, while keeping the enterprise application logic completely device-independent.

Keywords Web engineering · Architecture · Device independence · Dialog control

1 Introduction

With business processes becoming increasingly distributed in character and even beginning to exhibit mobile aspects, especially in areas such as field sales forces, logistics infrastructure etc. (Köhler and Gruhn 2004), users demand flexible access to many enterprise applications—ideally, any application should be available on any device, anywhere, anytime (Weiser 1993). These demands can virtually only be fulfilled by pursuing a thin-client approach (Sinha 1992), since economic considerations forbid implementing applications individually for every kind of device, and mobile devices typically have strict energy, memory, input and output limitations (Jing et al. 1999). Web-based front-ends seem to be ideal implementations of this concept, since the complete enterprise application logic resides on a central server, while the user interface (UI) consists entirely of web pages or similar renderings on client devices such as desktop PCs, PDAs, mobile phones etc. (Gaedke et al. 1998).

However, the I/O capabilities of these devices range widely, and characteristics like screen size do not only impact the page layout, but also affect how users work with an application (Butler et al. 2002). For example, a dialog that may be completed in a single step on a desktop browser may have to be broken up into multiple interaction steps on a mobile device whose small screen cannot accommodate large forms, as illustrated in Fig. 1: When creating a new user account for e.g. a travel agency's web portal, the user's address, travel preferences and desired password can be entered into a large one-page form in a desktop browser, but may have to be spread over multiple smaller pages on a mobile device, requiring the application to handle more interaction steps.

At first glance, this may seem like a simple issue that should be solvable relatively easily in an application that properly separates the implementation of UI and application logic, as suggested by the MVC pattern (Krasner 1988). While this separation is easily implemented in traditional window-based applications, web-based applications pose more of a challenge: The page-based pull communication

The Chair of Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

M. Book (✉) · V. Gruhn
Chair of Applied Telematics/e-Business,
Department of Computer Science, University of Leipzig,
Klostergasse 3, 04109 Leipzig, Germany
e-mail: book@ebus.informatik.uni-leipzig.de

V. Gruhn
e-mail: gruhn@ebus.informatik.uni-leipzig.de

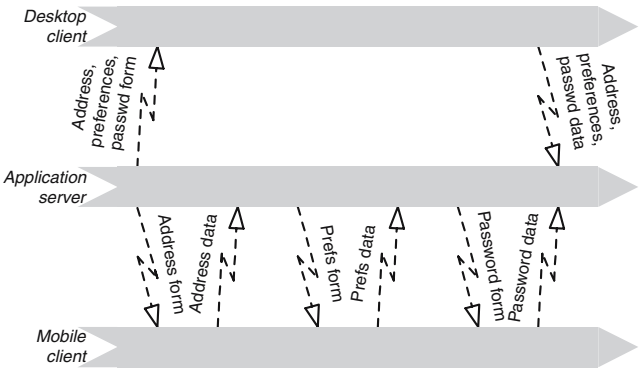

Fig. 1 Dialog flows on different devices

that is employed on the web requires additional logic to govern the sequence of pages displayed on the client and processing steps performed by the server. Often (e.g. when employing the commonly used Apache Struts framework, Apache Software Foundation, n.d.), this dialog control logic is not implemented explicitly and separately from the presentation or application logic, but intermingled with the latter's implementation. Consequently, any issues regarding the dialog sequence (such as different interaction patterns on different devices) need to be addressed in the application logic that was actually supposed to be device-independent.

A necessary, but not yet sufficient step towards resolving this problem is to employ a distinct dialog controller that receives requests from the UI web pages, dispatches them to the appropriate application logic and determines which UI page to send as a response, depending on the results returned by the application logic (Singh et al. 2002). Thereby, the dialog controller decouples the application and presentation logic, providing architectural support for device independence.

Device independence, however, can seldomly be implemented throughout the system: Except for very simple applications, the presentation logic is almost always device-specific since it contains or is responsible for generating the UI pages that will be delivered to the various devices. The second building block of a device-independent application must therefore be a means for linking the device-specific UI pages with the application logic while preserving the latter's device independence. These links can be established by a specification of the dialog flow defining the possible sequences of UI pages and processing steps that users encounter when interacting with the application using various devices. Given a suitable dialog controller, the dialog flow can then be controlled at run-time according to the specification instead of having to be hard-wired into the presentation, dialog control or application logic.

Thus, developers need a means for specifying the similarities and peculiarities of different devices' dialog flows. The specification should be compact and efficient in order to reduce development and maintenance effort, especially in software development projects following an agile process model where the specification is continually revised and expanded.

In this paper, we first present a concept for the non-redundant specification of dialog flows on multiple devices and show how it can be expressed in a graphical and XML-based notation. Next, we show how such dialog flows can be interpreted at run-time by a dialog control framework. Finally, we introduce a software process model for the dialog-driven development of multi-channel web applications, and discuss experiences gained from applying this method to the development of a web-based application front-end tailored to three device types. We conclude with an overview of ongoing and related work and discuss opportunities for further research.

2 Device-specific dialog specification

When building a web-based front-end for an enterprise application, the developers' goal should be to provide the same functionality across all devices, as demanded by the W3C's device independence principles (Gimson 2003). Specifically, the application's features should be structured similarly on all devices to help users with building a consistent conceptual model of the application, regardless of the device. This consistency with user expectations, also demanded by the ISO dialog principles (International Organization for Standardization 1996), can be supported by employing similar navigation schemes and interaction patterns on all devices. A consistent dialog flow throughout the application should also reduce development and maintenance effort as opposed to highly device-specific flows.

The above considerations call for dialog flows that are as generic as possible. Nevertheless, there will likely be parts of the dialog flow that cannot be specified device-independently but ask for specific solutions for one or more devices: Complex dialogs may require differing pagination on various devices, while selected features may be reduced or missing on some devices due to insufficient I/O capabilities, or because they are deemed unnecessary out of business process considerations. Meanwhile, other features may be exclusively available on certain devices for technical or process reasons (e.g. availability of location-based services on certain mobile phones).

When modeling the dialog flows of an enterprise application, the generic parts should only have to be modeled once, but still be applicable to all devices. In contrast, the specific parts should be modeled with explicit reference to the respective device. In this paper, we use the example of the Dialog Flow Notation (DFN) and Dialog Flow Specification Language (DFSL) to demonstrate how this approach can be supported by a notation.

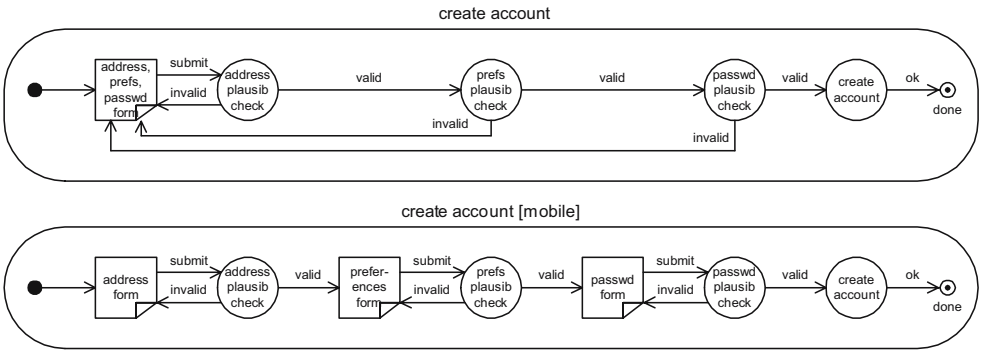

Fig. 2 Dialog module definition on generic and mobile presentation channel

2.1 Graphical notation

Inspired by Statecharts (Harel 1987), the Dialog Flow Notation (DFN) (Book and Gruhn 2004) models a web front-end's dialog flow as a transition network, i.e. a directed graph of states connected by transitions called a *dialog graph*. The notation refers to the transitions as *dialog events* and to the states as *dialog elements*, discerning atomic and compound elements.

Hypertext pages (symbolized by labeled dog-eared sheets and referred to by the more generic term *masks* here) and calls of application logic operations (symbolized by labeled circles and called *actions* from now on) constitute the *atomic dialog elements*. Every dialog element can generate and receive multiple events (symbolized by labeled arrows). Which element will receive an event depends both on the event and the generating element (e.g., an event e may be received by action A_1 if it was generated by mask M_1, but be received by action A_2 if generated by mask M_2). Events can carry parameters such as submitted form data or processing results, and thus facilitate communication between dialog elements.

In the DFN, dialog graphs are never free-standing, but always encapsulated in *dialog modules* that facilitate the connection and nesting of dialog graphs, as illustrated in the upper half of Fig. 2 using the example of a travel portal's *create account* module. Any module's dialog graph can contain sub-modules, and any module can itself be embedded in the dialog graphs of super-modules. When a module receives an event from the super-module that it is embedded in, traversal of its own dialog graph starts with the *initial anchor* (indicated by a thick dot). When its dialog graph terminates, its *terminal anchor* (indicated by a thin, circled black dot) generates a terminal event that is propagated to the super-module and continues the traversal of the dialog graph there. Note that the notation discerns between a module's *definition* (symbolized by a large box with rounded corners, the module name on top and its dialog graph inside, as in Fig. 2) and a module's *use* in a super-module's dialog graph (symbolized by a small oval with the module name inside, as shown by the use of the *create account* module in Fig. 3).

A module typically encapsulates one or more dialog masks, actions and possibly sub-modules implementing a certain functionality, process or behavior in the enterprise application (e.g. creating an account, logging in, searching for hotels, booking a room etc.). The modules' definitions are decoupled from each other, but through the interfaces of their initial and terminal events, they can call and return results to each other. Depending on the functionality realized by a module, it may only be used in one place (in a travel portal, e.g., a *book room* module will only be nested into the *accommodation* module), or it may be called from (i.e., nested into) different modules in the same application (e.g., a *login* module may be called from any module that requires user authentication). This facilitates easy reuse of dialog fragments and enables developers to model the complete dialog flow of an application with a set of dialog modules that are nested into and connected with each other.

By default, modules are device-independent: The dialog graphs they contain are executed identically on all devices.

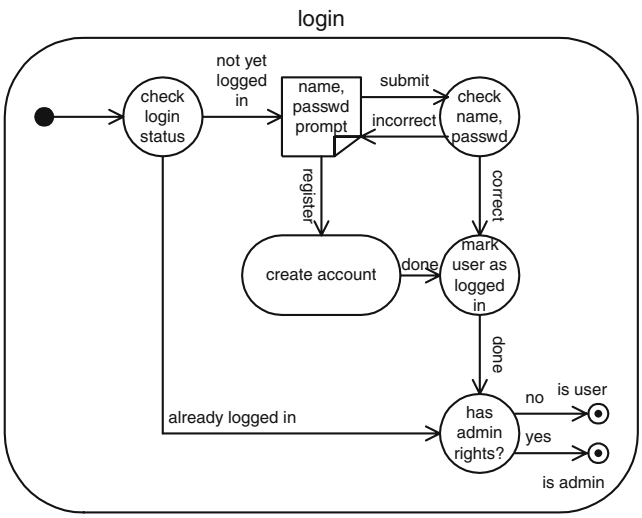

Fig. 3 Dialog module definition (*login* module) and use (*create account* module)

However, the DFN provides *presentation channels* as a construct that allows developers to specify different dialog flow variants for the same module. Usually, presentation channels represent certain device classes or markup languages. By specifying a *channel identifier* (noted in square brackets after the module name) for a module, the developer can restrict that dialog flow specification to a certain channel. It is then possible to specify multiple dialog module variants with the same name, but different dialog graphs, and associate them with different channels using different identifiers, as shown using the example of a mobile device channel for the *create account* module in the lower half of Fig. 2.

To keep the specification redundancy of this approach low, dialog modules have two additional semantics: Firstly, the generic definition of a module (i.e. the one without a channel identifier) is valid for all channels unless there are channel-specific module definitions (i.e. those with channel identifiers), in which case the specific definitions override the generic ones for the respective channels. For example, if an application supports three different channels for desktop browsers, PDAs and WAP-enabled mobile phones, and the dialog flow for the *create account* module shall be the same on all devices except those using the XHTML mobile profile, only two module definitions are necessary, as shown in Fig. 2: On the generic channel, we display one mask that prompts the user for all necessary data and then process that data in three consecutive actions on the server. In contrast, on the mobile channel, we display separate masks for each part of the form and process the data after each request.

This way, the notation not only saves the redundant specification of identical dialog flows, but also supports the convenient reuse of device-independent application logic: The four actions processing the user input can be reused on all channels in different dialog flows, yet need to be implemented only once.

Note that channel identifiers can only be given in modules' definitions, but not for their use—when nesting a dialog module into another, the channel identifier is never specified since it is either unknown or redundant: If the super-module has a generic dialog flow, we do not know at design-time through which channels users will traverse it at run-time, so we cannot specify a channel for its sub-modules. On the other hand, if the super-module has a channel-specific dialog flow, specifying a channel for its sub-modules is redundant because the sub-modules must obviously provide a dialog flow for the same channel. Either way, the actual dialog flow variant to use for the sub-module must be determined at run-time when the device employed by the user is known.

Figure 3 illustrates this mechanism using the example of a *login* module that prompts a user to log into the system (unless he is already logged in) and terminates with an event that indicates to the calling super-module if this user has regular or administrator privileges. Since the *login* module's simple dialog graph shall be the same on all devices, it was specified for the generic channel only. The *create account* module is nested into it and called when a user triggers the *register* event by following a corresponding link in the mask. It will then be decided at run-time if the *create account* module's generic or mobile-specific dialog graph variant needs to be traversed for this user's device.

A similar mechanism applies to the dialog masks' implementation: The DFN does not require developers to specify whether e.g. the desktop or mobile variant of a certain dialog mask shall be used on a certain channel, but relies on the dialog control logic to choose a suitable implementation at run-time, depending on the channel that a user's requests are coming in on. In contrast to dialog modules, however, there are obviously no "generic" mask implementations; rather, the developer needs to provide an implementation for each channel for the Dialog Control Framework's controller (described below) to choose from.

2.2 Specification language

The dialog flow semantics described in the preceding section already imply that some logic is required at run-time in order to choose suitable dialog flow variants for certain devices. We will present an example of such a dialog control logic in the following section, but first need a way to specify the dialog flows in machine-readable form. In case the dialog flows are specified in the DFN, we can use its associated XML-based Dialog Flow Specification Language (DFSL) for this purpose.[1]

In the conversion from DFN to DFSL, we can reduce the verbosity of the specification even further by eliminating partial redundancies within a module. In Fig. 2, for example, the last part of the *create account* module's dialog graph is the same on both channels (while the overlap is relatively small here, the dialog flow variants may share bigger parts in other examples). Instead of specifying the final *create account* action and *done* terminal event twice, it would be more efficient if we only had to specify it once. This is possible since the DFSL allows a more fine-grained specification of dialog flow variants than can be expressed in the graphical notation, as the following excerpt from a dialog flow specification document shows:

[1] Since the DFSL elements closely mirror the DFN elements, we will not introduce the language's syntax in detail here, but focus on the language constructs for presentation channel definition.

```
<dfs-flows>
  <in-module name="create account">
    <channel>
      <on-init>
        <call-mask>address, prefs, passwd form</call-mask>
      </on-init>
      ...
      <ex-action name="passwd plausib check">
        <on-event name="valid">
          <call-action>create account</call-action>
        </on-event>
        <on-event name="invalid">
          <call-mask>address, prefs, passwd form</call-mask>
        </on-event>
      </ex-action>
      <ex-mask name="create account">
        <on-event name="ok">
          <term-event>done</term-event>
        </on-event>
      </ex-mask>
    </channel>
    <channel name="mobile">
      <on-init>
        <call-mask>address form</call-mask>
      </on-init>
      ...
      <ex-action name="passwd plausib check">
        <on-event name="invalid">
          <call-mask>passwd form</call-mask>
        </on-event>
      </ex-action>
    </channel>
  </in-module>
  ...
</dfs-flows>
```

In this specification of the *create account* module, we first define the generic channel's dialog flow within the unnamed channel element, including all events generated by the *passwd plausib check* action and the final *create account* action. Next, we define the mobile channel's specific dialog flow within the channel element with the name="mobile" attribute—for example, the initial event leads to a different mask than on the generic channel, in accordance with the graphical specification in Fig. 2.

Note that the *valid* event of the *passwd plausib check* action, as well as the final *create account* action do not need to be specified on the mobile channel, since they were already defined on the generic channel. When interpreting the dialog flow specification, the dialog control logic implements the overriding semantics of the presentation channels by first looking events up in the channel-specific definition for the device that the user is employing. Only if an event is not defined in the channel-specific dialog flow, the dialog control logic looks it up in the generic definition instead.

3 Device-independent dialog control logic

After discussing requirements for a specification language that efficiently supports device independence by eliminating the redundancy that can be inherent in such dialog flow specifications, and introducing the Dialog Flow Notation and Dialog Flow Specification Language as examples, we still need to examine the dialog control logic that is required on the server in order to implement the semantics of these dialog flow specifications.

The dialog flow specification documents are parsed during the initialization of the web application, and the resulting dialog flow model comprising event tables for each module is continually used by the dialog control logic to look up the receivers of incoming events. Figure 4 shows the architecture of the Dialog Control Framework (DCF) (Book and Gruhn 2003) as an example of a dialog control logic implementing the device-independent specification constructs discussed in the previous section. In order to keep the actual dialog control logic completely device-independent, the requests coming in from the clients are received by *channel servlets* (step 1). Each of these servlets is tailored to the protocol and language specifics of a certain channel and knows how to retrieve the session identifier and parse the query string on that particular channel. This enables each servlet to extract the event data from its channel-specific encoding and build a dialog event object from it, which is then passed to the *dialog controller* (2). While the dialog event still contains information on the channel that it came in on, its structure is channel-independent.

To find the receiver of an incoming event, the dialog control logic must first establish the context in which this event was created. For this purpose, each user session is associated with a *module stack* that contains references to the nested modules that the user is currently navigating through. On top of the stack is the reference to the currently traversed module in the dialog flow model. Using the event's type, name and information on the generating element, the dialog controller can now look up the receiver for this event in the module's event tables. To implement the channel overriding semantics, the controller will first look for the event in the event tables of the channel-specific graph, and if it is not found, refer to the generic graph (3).

If the receiver found in the event tables is an action, the dialog controller will dispatch the event to that action (4), which may call back-end logic to update the applications' data model (5) and then returns a new event indicating the result of the operation (6). The dialog controller looks this event up in the dialog flow model again (7), where it may find that it leads to another action (in which case the cycle repeats) or to a module or mask. If the event receiver is a module, the dialog controller will push a reference to it onto the user's module stack in order to reflect the user's

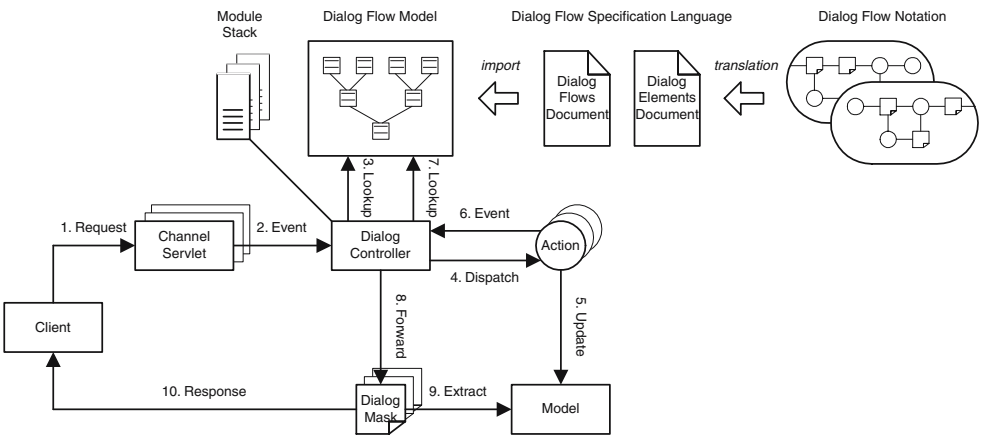

Fig. 4 Architecture of the Dialog Control Framework

updated position in the dialog flow, and then look up the receiver of that module's initial event. If the event receiver is a mask, the dialog controller will dispatch the request to the implementation for the corresponding presentation channel (e.g. a JavaServer Page, step 8), which can read information from the data model (9) to build a response that is finally sent back to the client (10).

Following the MVC paradigm, this architecture enforces a strict separation of presentation, application and dialog control logic since neither the masks nor the actions determine the next step in the dialog flow directly—rather, this decision is made by the dialog controller according to the specification provided by the developer. The channel-independent dialog control logic also simplifies the incorporation of additional devices: The developer only needs to implement a suitable channel servlet, specify any particular dialog flow variants for the new channel, and design an additional set of masks using a suitable markup language for the device, while the application logic and dialog control logic can remain unchanged.

4 Dialog-driven process model

In the following subsections, we introduce a dialog-driven process model (DDPM) that suggests how the DFN and DCF may be employed in the phases of a typical development process for a multi-channel web application. For clarity, we do not cover all tasks that have to be performed in each phase (such as conducting interviews to gather requirements, creating the screen design, specifying the database schema, etc.), but focus on the development of the dialog flow and associated deliverables (i.e. masks and actions) here. We illustrate the concepts presented for each phase with examples from the development of the ARGuS Travel Guide, a complex device-independent web application built using the DFN and DCF in order to examine the feasibility of this approach.

4.1 Requirements analysis phase

Early in the requirements analysis phase, a very coarse, high-level view of the application's dialog flow, showing just the relationships between the most important dialog modules, should be drafted in order to visualize the overall structure and scope of the project. This early draft can be derived from use cases and thus initiate the transition from informal requirements to a more formal and detailed specification of the application's look and feel.

The ARGuS Travel Guide, for example, bundles information on Leipzig's sights, restaurants, hotels, and public transportation schedules and makes it available through the web-based user interface of a portal system. Users can search for points of interest and save them in a personal travel planner to create their individual itinerary for a visit to the city. Most features of the system can be accessed either through a desktop browser, PDA or WAP-enabled mobile phone. Consequently, the first step in the dialog flow design was to identify the dialog modules that would later contain these use cases, without specifying their actual dialog graphs yet (Fig. 5).

4.2 Specification phase

The details of the dialog flow should then be worked out in the specification stage, preferably in incremental fashion: In order to ensure that the whole development process is driven by the users' needs, the coarse dialog graphs should be populated with an emphasis on dialog masks first, since these are the entities that will ultimately determine the user experience. The actions can be specified relatively coarsely at this early stage, and be refined in subsequent iterations. Note that it would be counterproductive to strictly separate the specification of masks and actions, since they work very closely together—however, in the early iterations, the masks should be considered as leading the specification, while the actions follow the requirements of the user interface.

For an application that serves multiple presentation channels, their dialog flows should be specified in parallel to ensure that the dialog structure is as similar as possible on all channels, allowing users to switch channels without having to rebuild their conceptual model of the application. In this case, giving preference to the masks during the specification and design phase is especially important to

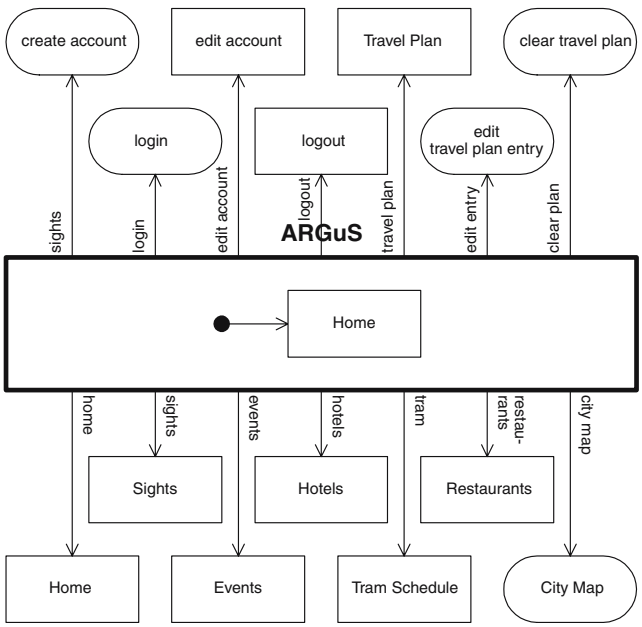

Fig. 5 High-level dialog structure of the ARGuS portal

prevent the design of the actions from becoming presentation channel-dependent. For example, to create a new account in the ARGuS portal, the user needs to provide quite a bit of data (address, travel preferences and desired password)—while this can be all acquired in one form on the desktop channel, we use a sequence of three pages on the mobile phone channel to cater to mobile devices' smaller screens (Fig. 6).

4.3 Design phase

By the end of the specification stage, the user experience should have been worked out in terms of which masks exist on various channels of the application and how the user can navigate between them. In the design stage, when the structure of the underlying application logic and data model is also designed, it is then time to refine the dialog graphs by inserting all necessary actions between the masks to process the users' input and prepare the system's output.

To complete the dialog graphs of the *create account* module, for example, we need to add logic for validating the data entered by the user. In order to keep this logic channel-independent, we implement it in three actions that are executed subsequently on the desktop channel, but interspersed with the masks on the mobile phone channel (as shown previously in Fig. 2). The necessity to distribute the input processing over several actions only becomes obvious if the channels are designed in parallel, and preference is given to the masks—had we designed for the desktop channel only, all the processing might have been implemented in one action that would have been unsuitable for reuse on the mobile phone channel. Therefore, if we want to avoid redundant implementation of application logic on multiple channels, the structure of the logic must be flexible enough to serve all channels—and the required degree of flexibility can only be gauged if the channels' user interfaces are specified first.

One might argue that finding the right granularity of masks and actions that is suitable for all presentation channels is a weak point of the notation, and indeed this may be a challenge if channels shall be added to an existing application at a later time. We will present an automated approach to solving this problem below.

As the application's data model matures in the design phase, this is also the time for specifying the data flow between the application logic and the user interface. This includes the detailed definition of the masks' contents, i.e. the data that the user should input and the system should output. This stage will likely be characterized by mutual feedback between the dialog flow specification and the application design: The dialog flow defines what the application logic should do in response to user interactions, guiding the design of the logic and data model, which in turn define the data structures that the masks and actions will process.

4.4 Implementation phase

In the implementation phase, the refined dialog flow model serves as essential input for the DCF, which will manage the users' interaction with the application accordingly. Since a machine-readable specification can be generated from the dialog flow model automatically, there is no need for developers to implement the specified dialog flows manually. Because of the modular nature of the dialog flow, modules may be added to the dialog flow model incrementally as the implementation of their constituent masks and actions progresses.

Obviously, since the dialog flow specification is continuously evolving from the first coarse sketches to the final detailed graphs, and the automatic conversion from DFN diagrams to executable DFSL documents encourages rapid prototyping, no clear lines can be drawn between the above phases. In practice, there will always be some degree of

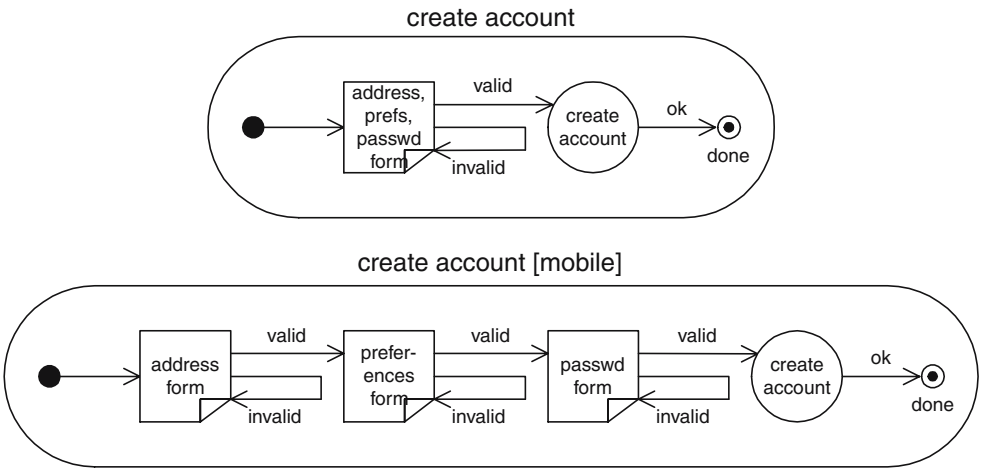

Fig. 6 Early *create account* dialog module sketches

overlap and iteration, depending on the type of application and the actual process employed.

4.5 Testing phase

In the testing phase, the DFN can be used on two levels: At first, developers can test the interaction of their masks and actions with the application logic on a relatively low level by embedding them into simple dialog graphs that merely serve as test drivers. Later, on a higher level, the complete dialog graph diagrams should serve as a reference to testers examining larger components or the whole system: By following all events in the diagrams, testers can verify that the dialog structure of the application was implemented as specified, does not contain any errors, and meets usability criteria.

Since the DFN encourages modularity in the dialog flow and the DCF enforces the separation of dialog flow specification, user interface design and application logic, any errors found in one of these tiers should be fixable without affecting other tiers or even other elements of the same tier. In contrast, the intermingling of application logic and dialog control logic in actions that is allowed by approaches such as Apache Struts (Apache Software Foundation, n.d.) bears the risk of introducing side effects when changing one aspect of the combined logic.

5 Project experience

To examine the feasibility of our approach to specifying and implementing web-based front-ends for device-independent enterprise applications, a team of three students developed the ARGuS travel guide that can be accessed using various devices. In Fig. 7, for example, users are searching for churches in Leipzig on the desktop, PDA and mobile phone channel (the latter splitting the search form and results list up into two masks).

During the specification phase, the system's dialog flows were defined in the DFN only for the desktop and mobile phone presentation channels at first (a PDA channel was not planned at this time). The dialog flow specifications underwent a number of iterations in order to arrive at a specification that reused as many actions on both channels as possible. The dialog flow diagrams were then translated into DFSL documents manually (a cumbersome process at the time, which has meanwhile been automated by a DFN modeling plug-in for the Eclipse IDE that creates DFSL documents automatically). In the implementation phase, the masks for the desktop presentation channel and all actions were implemented incrementally.

After a successful system test using the desktop channel, the masks for the mobile phone channel were implemented. Since the dialog flow for the mobile phone channel was completely planned earlier in coordination with the desktop channel, no additional application logic had to be implemented anymore. Close to the end of the project, the developers decided to add a channel for PDAs, a feature that was not part of the original requirements and had not been considered throughout the project. In contrast to the desktop channel that relied on rich HTML with layout and graphics, the PDA channel should serve light, virtually text-only HTML to the devices. Since the differences between the desktop and PDA channel were only in dialog mask markup, but not in pagination, the dialog flows for the desktop channel could be reused completely as a generic presentation channel. The specific dialog flows for the mobile phone channel were specified as extensions of the generic channel using the notation and framework's channel override mechanism. While some features, such as write access to the itinerary, were disabled in the mobile version out of usability considerations, other dialogs were split up into multiple masks to accommodate the smaller screen.

Using this approach, our development team required the following amounts of time to complete the implementation of each channel:

- *Desktop channel:* 2 months (implementation of application logic, 29 actions and 24 rich HTML masks)
- *Mobile phone channel:* 2 days (implementation of 22 WML masks)
- *PDA channel:* 1 day (extension of dialog flow specification, implementation of 24 light HTML masks)

The implementation phase was preceded by a 2-month specification and design phase that also included the definition of the dialog flows for the desktop and mobile phone channel.

Obviously, these implementation times cannot be generalized—more empiric evidence and a comparison to development projects that do not use the approach suggested here is necessary to draw valid conclusions on the efficiency of this method vs. others. While these numbers do not tell how much of the time saved for the implementation of the mobile and PDA channel was due to the non-redundant dialog flow specification for those channels, the initial results look promising and show that it is feasible to build complex web-based applications for multiple devices using our approach.

The numbers also reflect what would be intuitive expectations for the development effort of these channels: Since the rich and light HTML channels are very similar, one would expect the effort for adding the PDA channel to be quite low. The observed implementation effort of just 1 day confirms this and suggests that the notation and framework did not introduce any additional overhead. For

Fig. 7 Searching for sights on the desktop, PDA and mobile phone channel of ARGuS

the implementation of the mobile channel, the observed implementation time of 2 days is a bit higher than for the PDA channel, but still lower than expected. We consider this a payoff of the preceding 2 months of specification and design, where the dialog flows were iteratively revised up to a maturity that later actually enabled the developers to simply implement the WML masks without having to deal with any application logic in order to obtain a working mobile channel.

6 Automated content adaptation

As mentioned earlier, a weak point in the approach presented above may be the fine granularity of actions that is required to reuse them flexibly in dialog flows on different presentation channels (this especially concerns actions responsible for processing user input submitted through forms): The finer the actions are grained, the easier it is to adapt to different interaction patterns—however, very fine granularity also results in quite high specification, implementation and performance overhead. When specifying a dialog flow, the developer therefore needs to find a balance between the desired flexibility and the required granularity.

To solve this dilemma, we are currently exploring an approach that involves abstracting from concrete masks and actions, and just letting the developer specify which data the user shall be prompted for. Using XForms (World Wide Web Consortium 2003) in combination with the DFN, we can enable developers to specify both the dialog flow and the dialog contents in a mostly device-independent way. An extension to the framework can then generate the device-

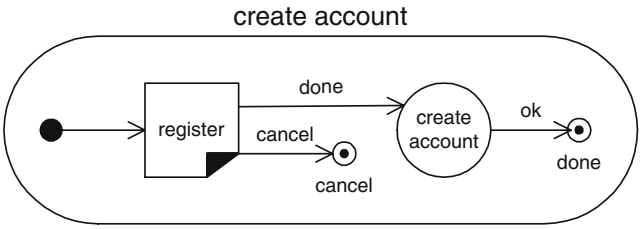

Fig. 8 Macro dialog graph of *create account* module with abstract *register* dialog, as specified by the developer

specific dialog masks and a suitable micro-dialog flow for obtaining and checking the user input automatically at run-time. These extensions to the DFN and DCF allow developers to focus their efforts on implementing the application logic and specifying the user interface, without having to deal with their adaptation to a broad spectrum of devices (Book et al. 2006).

In brief, we let the developer define the contents of each dialog in a so-called "abstract dialog specification," where we define a "dialog" as a collection of widgets that should all be displayed on the same hypertext page if a sufficiently large display is available (typically, one abstract dialog will correspond to one HTML page displayed on a desktop terminal, which may have to be broken up for smaller devices). The integration of these abstract dialog specifications with the DFN is quite straightforward: In the DFN's dialog graph diagrams, abstract dialogs are symbolized as sheets with a black dog ear (as opposed to concrete dialog masks, which are distinguished by a white dog ear) to convey their similarity to concrete masks: Abstract dialogs can be used in dialog graphs exactly the same way as concrete masks, as illustrated in Fig. 8. In this variation of the *create account* module, *register* is an abstract dialog that may be broken up into several dialog masks on some devices. Independently of the number of masks that it may be broken into, the register dialog will ultimately generate either a *done* or *cancel* event that determines if the account will actually be created.

When the extended DCF encounters an abstract dialog at run-time, it will parse the respective abstract dialog specification and generate the necessary concrete masks and micro dialog graphs connecting them automatically, as seen in the example of a two-step wizard in Fig. 9. We call the dialog graphs that are auto-generated to connect concrete dialog masks "micro dialog graphs" in order to distinguish them from the manually specified "macro dialog graphs" that the developer embedded the abstract dialog in. The dynamic generation of dialog masks and dialog graphs remains transparent for users, who will only be exposed to the familiar wizard interface in situations where an abstract dialog does not fit on a single concrete page.

The initial evaluation of a small-scale example indicated that this approach can decrease the volume of user interface-related code and specifications considerably, while incurring an acceptable increase in the execution time of the dialog control logic. Of course, the usability of the auto-generated dialog masks depends heavily on the availability of suitable profiles for recognizing the display characteristics of devices employed by users, the accuracy of the heuristics used to estimate the size of different widgets, and the layout algorithms employed to render masks in different markup languages. These areas still bear a lot of potential for optimization and evaluation, which is a prime topic of our ongoing research.

7 Related work

A number of notations for modeling user interface navigation have been proposed over time. However, many notations introduced specifically for the field of web-based UIs initially focused on data-intensive information systems

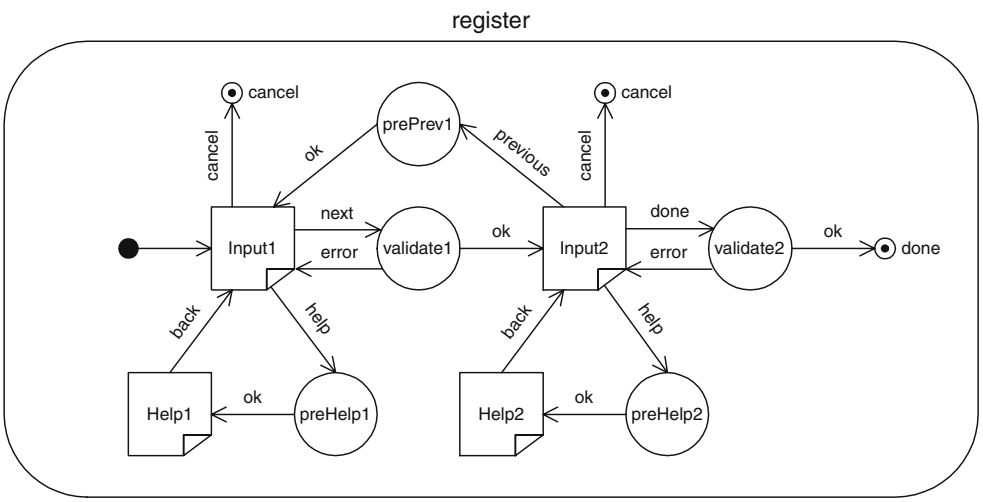

Fig. 9 Auto-generated micro dialog graph of *register* dialog with two concrete dialog masks

and only recently began to provide support for interaction-intensive web applications. The language WebML (Ceri et al. 2000), for example, is capable of modeling the layout and appearance of web pages independently of the output device using an abstract XML language for its presentation model, but does not seem to provide an overriding mechanism for the extension of generic dialog modules with channel-specific fragments, as implemented in the approach presented here. Similarly, the Web Composition Language (Gaedke et al. 1998) focuses on the specification of the dialog mask's contents, but does not distinguish between generic and channel-specific dialog flows.

More recent notations, often based on Statecharts (Harel 1987), also provide extensive support for interaction-intensive applications: For example, Leung et al. (Leung et al. 2000) use Statecharts to model dynamic web applications, but do not provide a means for specifying device-specific interaction patterns. The same is true for StateWebCharts (Winckler et al. 2004). While the HMBS model (de Oliveira et al. 2001) allows the channel-dependent specification of navigation patterns, it focuses on challenges such as synchronization that are introduced by multimedia elements embedded into hypertext. Schewe et al. (Schewe and Thalheim 2001) use a formal approach for modeling interaction and media objects that allows the specification of device-specific variants of media objects depending on the presentation channels' capabilities, but do not provide a means for the non-redundant specification of device-independent dialog flows. Last but not least, the formal model for web interactions proposed by Graunke et al. (Graunke et al. 2003) helps to identify and deal with unexpected situations in the dialog flow (e.g. backtracking), but does not address device independence issues. XForms (World Wide Web Consortium 2003) separates form contents from control flow, but does not offer constructs for modeling and controlling complex, modular dialog flows as the DFN does.

Most tools offering dialog control implementation support for web applications follow the Front Controller design pattern to facilitate easier dialog control. The Apache Project's Struts framework (Apache Software Foundation, n.d.) is the most popular solution today, however, it forces developers to combine application logic and dialog control logic in its actions: The Struts controller only decides which action should receive incoming requests, but the actions then decide which view to display next. Since the application logic is thus not completely decoupled from the dialog flow, reusing it on different channels is not always possible, making device-independent design cumbersome.

The challenges posed by different devices' interaction patterns are addressed in the Sisl (Several Interfaces, Single Logic) approach (Ball et al. 2000). Its "service monitor" can process unordered or incomplete input from a wide range of client devices. However, since it uses acyclic graphs to model dialogs, it is more suitable for simple linear and branched dialog structures than for highly interactive applications with nested and cyclic dialogs. The need to spread complex forms over multiple interaction steps on small-screen devices instead of presenting them as a whole is addressed by the Renderer-Independent Markup Language (RIML) (Ziegert et al. 2004), an extension of XHTML 2.0 which contains semantic information for an automatic pagination engine. Collecting the data fragments coming in from the split-up forms is the task of a proxy between the client and server in that approach. In contrast, our extension to the Dialog Control Framework described in the previous section enables it to manage the necessary micro-dialog flows directly. Finally, a number of approaches (e.g. by Ceri et al. 2004) place the responsibility for content adaptation on the client; however, we do not follow this approach since we aim to keep the client as "thin" as possible.

8 Discussion

In this paper, we discussed possible redundancies in the specifications of dialog flows for web-based front-ends for enterprise applications and suggested an approach for eliminating these redundancies in the specification. To illustrate this approach, we presented a graphical notation and XML-based specification language that allows the modular specification of generic, device-independent dialog flows that are extended by device-specific dialog flows where necessary. We also showed how these specifications can be used by a framework that controls an application's dialog flow by implementing the semantics of overriding generic flow specifications with more specific ones. Using this approach, developers can reuse the device-independent application logic across multiple devices that behave mostly similar to create a consistent user experience, yet may differ in certain dialog structures where dictated by their I/O capabilities.

The feasibility of this approach was demonstrated in the development of a complex web-based application serving three channels. The observed implementation times for these channels indicate that the detailed specification of the dialog flows and the complete separation of application, presentation and dialog control logic paid off by reducing the development effort for two of the three channels to the mere implementation of dialog masks. However, since the desktop and mobile dialog flows were designed in parallel, it is not possible to quantify how much effort went into which channel. It would be an interesting experiment to add a channel for a device with restricted I/O capabilities late in the project and observe the development effort this incurs.

Based on our experiences gained from the development of the ARGuS web application, we predict that a dialog-driven process using the Dialog Flow Notation and Dialog Control Framework can reduce the development effort and cost for web applications in a number of ways: Firstly, the development effort for application-specific dialog control logic is eliminated since the DCF that contains all this logic can be reused as a black box. Secondly, the redundant effort of first specifying an application's dialog flow in some notation and then manually implementing it is eliminated since the DFN/DFSL specifications serve as direct input to the DCF without the need for further programming. Thirdly, redundant development effort for multiple presentation channels is eliminated since the DFN and DCF encourage reuse of the device-independent application logic and provide means to reuse similar parts of the dialog flow across channels. Fourthly, the risk of late discovery of flaws in the user interface and business processes is reduced since the intuitively understandable DFN allows the involvement of non-technical stakeholders throughout the project, and the seamless transition from specification to implementation of dialog flows using the DCF allows rapid prototyping. We hypothesize that in combination, these effects will lead to a reduced overall cost of web application development and a shorter time to market, while at the same time providing developers with means to structure web applications' user interfaces on various devices in a more user-friendly way.

References

Apache Software Foundation (n.d.). Apache Struts Project. Retrieved from http://struts.apache.org.

Ball, T., Colby, C., & Danielsen, P. (2000). Sisl: Several interfaces, single logic. *International Journal of Speech Technology, 3*(2), 91–106.

Book, M., & Gruhn, V. (2003). A dialog control framework for hypertext-based applications. In H. Lin & H. Ehrich (Eds.), *Proceedings of the 3rd international conference on quality software (QSIC 2003)* (pp. 170–177). Los Alamitos, CA: IEEE Computer Society Press.

Book, M., & Gruhn, V. (2004). Modeling web-based dialog flows for automatic dialog control. In *19th IEEE international conference on automated software engineering (ASE 2004)* (pp. 100–109). Los Alamitos, CA: IEEE Computer Society Press.

Book, M., Gruhn, V., & Lehmann, M. (2006). Automatic dialog mask generation for device-independent web applications. In *Proceedings of the 6th International Conference on Web Engineering (ICWE 2006)* (pp. 209–216). New York: ACM Press.

Butler, M., Giannetti, F., Gimson, R., & Wiley, T. (2002). Device independence and the web. *IEEE Computing, 6*(5), 81–86 (Sep–Oct).

Ceri, S., Dolog, P., Matera, M., & Nejdl, W. (2004). Model-driven design of web applications with client-side adaptation. In *Proceedings of the 4th International Conference on Web Engineering (ICWE 2004), Lecture Notes on Computer Science, 3140*, 201–214.

Ceri, S., Fraternali, P., & Bongio, A. (2000). Web modeling language (WebML): A modelling language for designing web sites. *Computer Networks, 33*, 137–157.

de Oliveira, M. C. F., Turine, M. A. S., & Masiero, P. C. (2001). A statechart-based model for hypermedia applications. *ACM Transactions on Information Systems, 19*(1), 28–52.

Gaedke, M., Beigl, M., Gellersen, H., & Segor, C. (1998). Web content delivery to heterogeneous mobile platforms. In *Advances in Database Technologies, Lecture Notes in Computer Science, vol. 1552*.

Gimson, R. (2003). Device independence principles, W3C working group note 01 September 2003. Retrieved from http://www.w3.org/TR/2003/NOTE-di-princ-20030901/, Sep.

Graunke, P., Findler, R., Krishnamurthi, S., & Felleisen, M. (2003). Modeling web interactions. In *Proceedings of the 12th European symposium on programming, lecture notes on computer science, vol. 2618* (pp. 238–252). Berlin Heidelberg New York: Springer.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8*(3), 231–274.

International Organization for Standardization (1996). Ergonomic requirements for office work with visual display terminals (VDTs)—Part 10: Dialogue Principles. *ISO*, 9241–10.

Jing, J., Helal, A., & Elmagarmid, A. (1999). Client-server computing in mobile environments. *ACM Computing Surveys, 31*(6), 117–157 (Jun).

Köhler, A., & Gruhn, V. (2004). Analysis of mobile business processes for the design of mobile information systems. In K. Bauknecht, M. Bichler, & B. Pröll (Eds.), *E-commerce and web technologies* (p. S. 238–247). Berlin Heidelberg New York: Springer.

Krasner, G. (1988). A cookbook for using the model-view-controller user interface paradigm in smalltalk. *Journal of Object-oriented Programming, 1*(3), 26–49.

Leung, K., Hui, L., Yiu, S. M., & Tang, R. (2000). Modeling web navigation by statechart. In *Proceedings of the 24th annual international computer software and applications conference (COMPSAC 2000)*. Los Alamitos, CA: IEEE Computer Society Press.

Schewe, K.-D., & Thalheim, B. (2001). Modeling interaction and media objects. *Proceedings of the 5th International Conference on Applications of Natural Language to Information Systems, Lecture Notes in Computer Science, 1959*, 313–324.

Singh, I., Stearns, B., & Johnson, M. (2002). *Designing enterprise applications with the J2EE platform (2nd ed.)*. Reading, MA: Addison-Wesley.

Sinha, A. (1992). Client-server computing. *Communications of the ACM, 35*(7), 77–98 (Jul).

Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Communications of the ACM, 36*(7), 75–84 (Jul).

Winckler, M., Barboni, E., Farenc, C., & Palanque, P. (2004). SWCEditor: A model-based tool for interactive modeling of web navigation. In *Proceedings of the 5th international conference on computer-aided design of user interfaces (CADUI 2004)* (pp. 55–66). Dordrecht: Kluwer.

World Wide Web Consortium (2003). XForms 1.0, W3C recommendation. Retrieved from http://www.w3.org/TR/2003/REC-xforms-20031014/, Oct.

Ziegert, T., Lauff, M., & Heuser, L. (2004). Device independent web applications—The author once—Display everywhere approach. *Proceedings of the 4th International Conference on Web Engineering (ICWE 2004), Lecture Notes in Computer Science, 3140*, 244–255.

Matthias Book is a doctoral candidate at the Deutsche Telekom Chair of Applied Telematics/e-Business at the University of Leipzig. His research interests are in the specification and control of dialog and data flows in web-based applications, with a special focus on designing for device independence. He is a co-author of over 25 refereed publications in international software and web engineering conferences and journals.

Volker Gruhn is a full professor and holder of the Deutsche Telekom Chair of Applied Telematics/e-Business at the University of Leipzig. His research interests are in agile model-driven development, especially in methods for the development of mobile, distributed software systems. He is author and co-author of about 120 national and international publications. In the 1990s, Volker Gruhn worked at the Fraunhofer Institute for Software and Systems Engineering (ISST) and was a member of the executive board of a software company of Veba AG. He subsequently became professor for applied computer science at the University of Dortmund, with a research focus on the development of component-based software architectures and e-business applications. He is founder and chairman of the board of software company adesso AG.