Engineering Context-Adaptive UIs for Task-Continuous Cross-Channel Applications

Enes Yigitbas, Stefan Sauer

▶ To cite this version:

Enes Yigitbas, Stefan Sauer. Engineering Context-Adaptive UIs for Task-Continuous Cross-Channel Applications. 6th International Conference on Human-Centred Software Engineering (HCSE) / 8th International Conference on Human Error, Safety, and System Development (HESSD), Aug 2016, Stockholm, Sweden. pp.281-300, 10.1007/978-3-319-44902-9_18 . hal-01647701

HAL Id: hal-01647701
https://inria.hal.science/hal-01647701

Submitted on 24 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Engineering Context-Adaptive UIs for Task-Continuous Cross-Channel Applications*

Enes Yigitbas and Stefan Sauer

Paderborn University, s-lab - Software Quality Lab,
Zukunftsmeile 1, 33102 Paderborn, Germany
{eyigitbas,sauer}@s-lab.upb.de

Abstract. The user interfaces (UIs) of interactive systems become increasingly complex since many heterogeneous and dynamically changing contexts of use (platform, user, and environment) have to be supported. Developing UIs for such interactive systems often requires features like UI adaptivity and seamless task-continuity across devices, demanding for sophisticated UI development processes and methods. While existing engineering methods like human-centered design process and model-based UI development approaches serve as a good starting point, an integrated engineering process addressing specific requirements of adaptive UIs supporting task-continuity across different devices is not fully covered. Therefore, we present a model-based engineering approach for building context-adaptive UIs that enable a personalized, flexible and task-continuous usage of cross-channel applications. Our engineering approach supports modeling, transformation and execution of context-adaptive UIs. To show the feasibility of our approach, we present an industrial case study, where we implement context-adaptive UIs for a cross-channel banking application.

Keywords: Model-based Development, UI Adaptation, Multi-Device UI Development, Cross-channel Applications, Task-Continuity

1 Introduction

Today users are surrounded by a broad range of networked interaction devices (e.g. smartphones, smartwatches, tablets, terminals etc.) for carrying out their everyday activities. Due to the growing number of such interaction devices, new possible interaction techniques (e.g. multi-touch or tangible interaction) and distributed user interfaces transcending the boundaries of a single device, software developers and user interface designers are facing new challenges. As the user interfaces of interactive systems become increasingly complex since many heterogeneous contexts of use (platform, user, and environment) have to be supported, it is no longer sufficient to provide a single "one-size-fits-all" user interface. The

* This work is based on "KoMoS", a project of the "it's OWL" Leading-Edge Cluster, partially funded by the German Federal Ministry of Education and Research (BMBF).

problem increases even more if we consider dynamic changes in the context of use. In this case, allowing flexible and natural interaction with such devices requires additional features like UI adaptivity to automatically react to the changing context of use parameters at runtime and task-continuity for supporting a seamless handover between different devices. For illustrating the problem, we introduce a real world example scenario which is derived from the banking domain.

While customers accessed banking services solely via isolated channels (through banking personnel or ATM) in the past, using different channels during a transaction is nowadays increasingly gaining popularity. Depending on the situation, customers are able to access their banking services where, when and how it suits them best. In the world of Omni-Channel-Banking, customers are in control of the channels they wish to use, experiencing a self-determined "Omni-Channel-Journey". For example, if the customers pursue an "Omni-Channel-Journey" for a payment cashout process, they can begin an interaction using one channel (prepare cashout at desktop at home), modify the transaction on their way on a mobile channel, and finalize it at the automatic teller machine (ATM) (see figure 1). It is important to notice, that each channel has its own special context of use and eventually the contextual parameters (user (U), platform (P), and environment (E)) can change if there is a channel switch. Thus, Omni-Channel-Banking brings the industry closer to the promise of true contextual banking in which financial services become seamlessly embedded into the lives of individual and business customers.

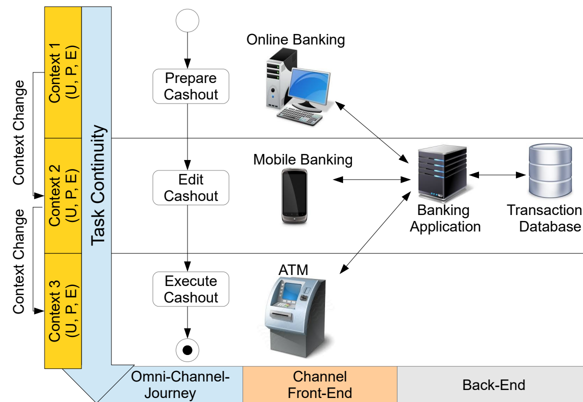

Fig. 1. Example Scenario: Omni-Channel-Journey with Task-Continuity

However, the advancement from Multi-Channel- to Omni-Channel-Banking (compare table 1) is a difficult task for developers of such systems. Developers are facing the following challenges:

Multi-Channel-Banking	**Omni-Channel-Banking**
Fixed channel usage	Flexible channel usage
Separation of channels	Integration of channels
Data redundancy in channels	Data synchronization between channels
Little or no channel switch	Continuous channel switch

Table 1. Multi-Channel-Banking vs. Omni-Channel-Banking

- **C1:** Support for modeling and adaptation of heterogeneous user interfaces (UIs) satisfying different contexts of use (user, platform, environment).
- **C2:** Support for a flexible channel usage depending on the context.
- **C3:** Support for a seamless handover between channels allowing task-continuity. When the user moves from one device to another, the user is able to seamlessly continue her task.

According to Petrasch [1], the effort of implementing an application's user interface constitutes at least 50 percent of the total implementation effort. Developing separate applications for each potential device and operating system is neither a practical nor a cost effective solution, especially if we consider heterogeneous contexts of use as they were described in the example scenario above. Model-based User Interface Development (MBUID) is a promising candidate for mastering the complex development task in a systematic, precise and appropriately formal way. For tackling the above mentioned challenges we present a model-based engineering process for context-adaptive UIs which integrates human-centered design aspects into the development process. Our engineering approach provides specific support especially for modeling, transformation, and execution of context-adaptive UIs enabling task-continuous usage of cross-channel applications.

The paper is structured as follows: First, we describe some background information and related work in the area of engineering methods for UIs, covering multi-device and cross-channel UI development as well as UI adaptation. Then, we present our model-based engineering approach for the development of context-adaptive UIs supporting task-continuity. After that, we present the instantiation of our engineering approach based on a case study from the banking domain. Finally, we conclude with a summary and an outlook for future research work.

2 Background and Related Work

In recent years, a number of approaches have addressed the problem of engineering user interfaces for different contexts of use. Our work is inspired by and based on existing approaches from the area of model-based UI development, adaptive UIs and distributed user interfaces (DUIs). In this section, we especially review prior work that explores the development of multi-device, cross-channel and adaptive user interfaces (UIs) supporting task-continuity.

2.1 Multi-device UI development

The development of multi-device UIs has been subject of extensive research [2], where different approaches were proposed to support efficient development of UIs for different target platforms. On the one hand, model-based UI development approaches were proposed which aim to create multi-device UIs based on the transformation of abstract user interface models to final user interfaces. Widely studied approaches are UsiXML [25], MARIA [3] and IFML[1] that support the abstract modeling of user interfaces and their transformation to multi-device UIs including web interfaces. In [4], we present a specialized approach for model-based development of heterogeneous UIs for different target platforms including self-service systems like ATMs. On the other hand, there are also existing approaches like Damask [5] and Gummy [6] following the WYSIWYG paradigm. While Damask is a prototyping tool for creating sketches of multi-device web interfaces, Gummy is a design environment for graphical UIs that allows designers to create interfaces for multiple devices using visual tools to automatically generate and maintain a platform-independent description of the UI. While above mentioned approaches support the development of multi-device UIs regarding specification and generation of UIs for different target platforms, they do not cover mechanisms to support channel switches and data synchronization between different target platforms at runtime.

2.2 Cross-channel UI development

Previous work by the research community has covered concepts and techniques to dynamically support the distribution of UIs by supporting task-continuity for the end-users. One of the concepts is called *UI migration*, which follows the idea of transferring a UI or parts of it from a source to a target device, while enabling task-continuity through carrying the UI's state across devices. In [9], we present a model-based framework for the migration and adaptation of user interfaces across different devices. In [7] and similarly in [24], the authors present a solution to support migration of interactive applications among various devices, including digital TVs and mobile devices, allowing users to freely move around at home and outdoor. The aim is to provide users with a seamless and supportive environment for ubiquitous access in multi-device contexts of use. In the case of web applications, most solutions rely on HTML proxy-based techniques to dynamically push and pull UIs [8]. An extension of this concept is presented in [10], where the authors propose XDStudio to support interactive development of cross-device UIs. In addition, there is also existing work on the specification support for cross-device applications. In [11] for example, the authors present their framework Panelrama which is a web-based framework for the construction of applications using DUIs. In a similar work [12], the authors present Conductor, which is a prototype framework serving as an example for the construction of cross-device applications. While above mentioned approaches support the specification and development of cross-channel UIs for different target platforms, they

[1] http://www.ifml.org

do not address combining the aspect of UI adaptation for different contexts of use with task-continuity.

2.3 Adaptive UIs

In recent research works, adaptive UIs have been promoted as a solution for context variability due to their ability to automatically adapt to the context of use at runtime. A key goal behind adaptive UIs is plasticity, denoting a UIs ability to preserve its usability across multiple contexts of use [13]. Norcio and Stanley [14] consider that the idea of an adaptive UI is straightforward since it simply means: "The interface should adapt to the user; rather than the user must adapt to the system." Based on [15] we can generally differentiate between the following types of adaptive UIs:

Adaptable user interfaces allow interested stakeholders to manually adapt the desired characteristics; example: a software application that supports the manual customization of its toolbars by adding and removing buttons. *Semi-automated adaptive user interfaces* automatically react to a change in the context-of-use by changing one or more of their characteristics using a predefined set of adaptation rules. For example: an application can use a sensor to measure the distance between the end-user and a display device, and then trigger predefined adaptation rules to adjust the font-size. *Fully-automated adaptive user interfaces* can automatically react to a change in the context-of-use. However, the adaptation has to employ a learning mechanism, which makes use of data that is logged over time. One simple example is a software application, which logs the number of times each end-user clicks on its toolbar buttons and automatically reorders these buttons differently for each end-user according to the usage frequency.

A classification of different adaptation techniques was introduced by Oppermann [18] and refined by Brusilovsky [17]. UIs with adaptation capabilities have been proposed in the context of various domains (e.g. [19],[20],[26] or [27]) and there are also proposals for integrating adaptive UI capabilities in enterprise applications (e.g. [16]). Although above mentioned approaches already present technical solutions for supporting UI adaptivity and model-based development approaches were proposed in the past (e.g. [23]), an engineering process for the development of context-adaptive UIs enabling task-continuity is not fully covered. Leaning on existing concepts of adaptive/cross-channel UIs and our previous work [21], where we propose a meta-method for engineering advanced user interfaces, we present a model-based engineering approach for developing context-adaptive UIs enbaling task-continuous usage of cross-channel applications.

3 Engineering Process

In this section, we present a model-based engineering process for development of context-adaptive UIs in order to tackle the motivated challenges C1, C2 and C3. Figure 2 gives a general overview of our engineering process which is divided

into four main steps:: *UI Modeling*, *Adaptation Modeling*, *Transformation*, and *Execution and Adaptation*.

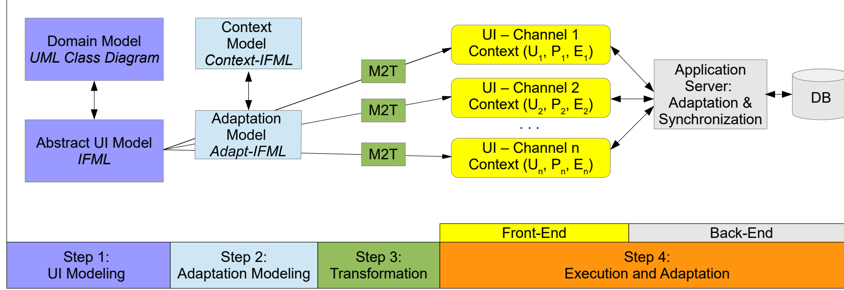

Fig. 2. Model-based engineering approach for developing context-adaptive UIs supporting task-continuity

In the following subsections each engineering step is explained in more detail by describing the used artifacts and activities.

3.1 UI Modeling

For supporting the development of various UIs allowing access for different channels and minimizing recurrent development efforts in establishing the needed Front-ends, our engineering process begins with the *UI Modeling* step. The goal of this step is specification of an abstract UI representation that serves as a basis for the transformation and generation of heterogeneous UI variants for different channels. For accomplishing this step, we decided to use existing modeling languages standardized by the Object Management Group (OMG). As a consequence, in the *UI Modeling* step we make use of *UML Class Diagrams* and the *Interaction Flow Modeling Language (IFML)*. In the beginning of the engineering process, the developers use *UML Class Diagrams* for specifying a *Domain Model* describing the data entities of the user interface. After that, they make use of the *Interaction Flow Modeling language (IFML)* that supports the modeling of the structure, content and navigation needed to characterize the UI Front-End in an abstract manner. For performing the *UI Modeling* step, the developers are able to use the open source IFML Editor Eclipse plugin[2] that enables a complete specification of the *abstract UI model* referencing the *Domain Model* entities.

3.2 Adaptation Modeling

After modeling the core UI characteristics using a *Domain* and *Abstract UI* model, the developers need to specify adaptation rules that decide how the user

[2] http://ifml.github.io

interface is adapted to specific contextual parameters (e.g. user, platform and environment) at runtime. Therefore, our engineering approach provides an *Adaptation Modeling* step, which allows explicit modeling of UI adaptation rules. For defining the adaptation rules, we first introduce a metamodel for context modeling (*Context-IFML*) that serves as a basis for describing the contextual parameters in a fine grained and extensible manner. After that, we introduce an adaptation metamodel (*Adapt-IFML*) for specifying complete adaptation rules.

Figure 3 gives an overview of our metamodel for context modeling. The metamodel *ContextModel* (*Context-IFML*) consists of the three main classes *User*, *Platform*, and *Environment* that characterize the specific contextual parameters. The class *User* for example, is subclassified in *PersonalInformation*, *Preferences* and *Knowledge*, while the classes provide different attributes to specify varying peculiarities of a user. Similarly different forms of the platform and characteristics of the environment can be specified with the help of *Context-IFML*. This way, the developers are able to specify heterogeneous context of use scenarios and to explicitly define the events and conditions for adapting the UI. For the complete definition of an adaptation rule we introduce the metamodel for adaptation modeling (*Adapt-IFML*) that is depicted in figure 4. Regarding our metamodel an adaptation model consists of different adaptation rules that are structured according to the Event-Condition-Action (ECA) paradigm. While the preconditions of an adaptation rule (event and condition) are specified based on the context model, the *Action* class provides different mechanisms to adapt the UI at runtime. Basically three different UI adaptation operations are supported, namely *Task-*, *Navigation-* and *Layout-ChangeOperation*, while each of them can be aggregated to more complex UI adaptation operations called *ComposedAction*. *Task-ChangeOperations* support adaptation by flexibly showing and hiding interaction elements of the UI. By using the operations *AddViewComponent* or *DeleteViewComponent* specific UI interaction elements like tables, textfields etc. can be shown or hidden depending on the specified adaptation rule. In a similar way, the *Navigation-ChangeOperation* can be used for adding, deleting and redirecting links between user interface flows. This way, the navigation flow of the UI can be flexibly adapted based on the contextual parameters that are defined in the preconditions. Our metamodel for adaptation modeling also supports the definiton of *Layout-ChangeOperations* like *ChangeFont* or *SplitViewContainer* for dividing a complex UI view container into multiple view containers, so that small screen sizes for example are satisfied. For supporting the *Adaptation Modeling* step, we extended the standard IFML metamodel with *Context-IFML* and *Adapt-IFML* according to the previous description. Based on this extensions the developers are able to specify different UI adaptation rules that are evaluated at runtime.

3.3 Transformation

After finishing the modeling steps, the developers are facing the task to define transformations to generate numerous UI Front-Ends for different channels. Therefore, in the transformation step several model-to-text transformation

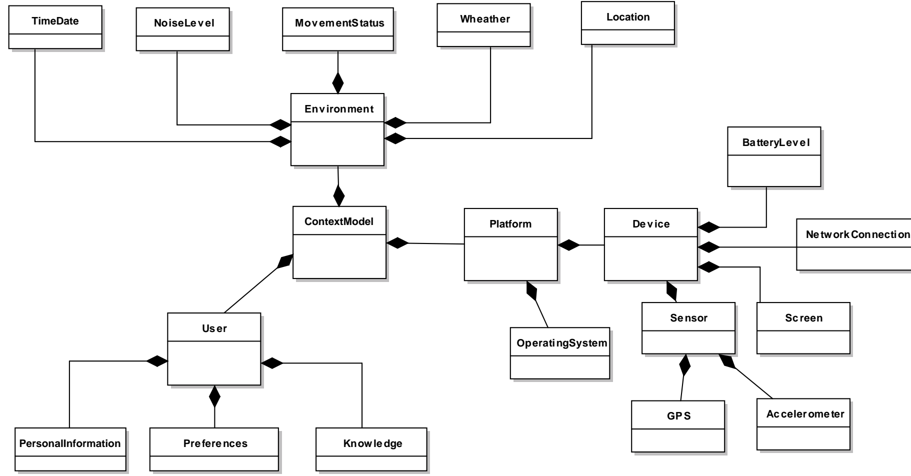

Fig. 3. Metamodel for Context Modeling (Context-IFML)

(M2T) templates are defined that transfer the abstract UI models into final UIs that are running on different target platforms in order to support access to different channels (Front-End). By generating different UI views and supporting different UI - Channels, users are able to flexibly select the channel of their choice depending on the context. For supporting the transformation step, we implemented an Xtend[3] plugin that maps the abstract UI model elements to specific elements of a target language/technology. The implemented Xtend plugin includes different Xtend templates to transfer the IFML source model into final user interfaces that support different context of use parameters. During the transformation process, not only the mapping between the source abstract UI model and the target final user interface (FUI) is established, but also the specified adaptation rules are transformed into the target language of the FUI, so that the generated UI can adapt itself at runtime according to the predefined rules.

3.4 Execution and Adaptation

For practical usage of the resulting final user interfaces that were reached after the modeling and transformation steps, the developers need an execution environment that executes and adapts the UIs according to the specified adaptation rules. For supporting this task, we present a solution architecture enabling automatic UI adaptation and synchronization. Figure 5 gives an architectural overview of the execution environment for context-adaptive UIs. The solution architecture is based on IBM's MAPE-K [22] loop, where an *Adaptation Manager* monitors and adapts the *Context-Adaptive UI* that can run on different

[3] http://www.eclipse.org/xtend

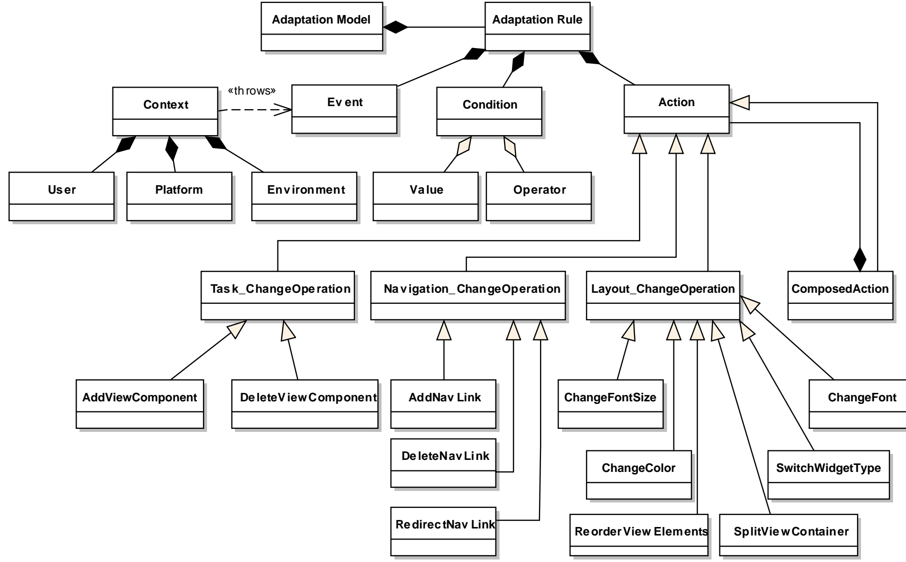

Fig. 4. Metamodel for Adaptation Modeling (Adapt-IFML)

platforms. The *Adaptation Manager* consists of five main components that work according the adaptation rules that were specified in the *Adaptation Model*: The *Monitor* component is responsible for observing context information that are provided by the *Context Manager*. All contextual parameters that were defined based on the *Context Model* are observed, so that the *Analyze* component is able to decide whether adaptation is needed. Therefore, the conditions for triggering an adaptation rule are analyzed and if adaptations are required, an adaptation schedule is done by the *Plan* component. Finally, the adaptation operations are performed by the *Execute* component, so that an adapted UI can be presented. The *Knowledge* base is responsible for storing data that is logged over time and can be used for inferring future adaptation operations. In addition to UI adaptation, for supporting a seamless handover between channels and allowing task-continuity for the user, our solution architecture includes a dedicated *Synchronization Server* which is responsible for storing and sharing of data (e.g. UI state or user preferences). The UI state including entered input data by the users is stored and restored so that when the user moves from one channel to another, the user is able to seamlessly continue her task on the new UI-Channel.

4 Instantiation of Engineering Process

In this section, the instantiation of the development process according to the previously described engineering process is presented in more detail. To show the feasibility of our approach, we first present the setting of an industrial case-study

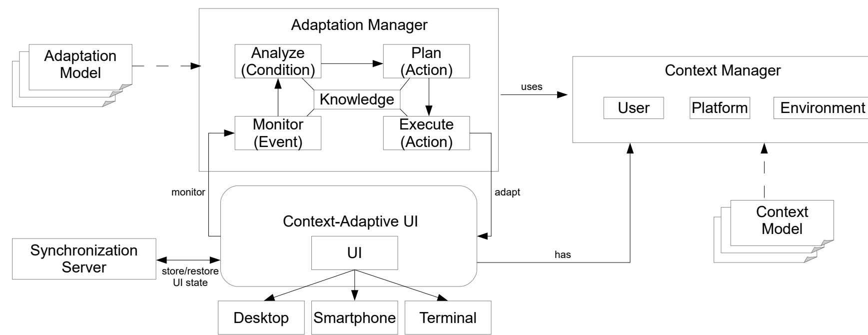

Fig. 5. Architectural overview of context-adaptive UIs

dealing with the implementation of context-adaptive UIs for a cross-channel banking application employing web-based technologies. After that, we present the instantiation of the engineering process by describing the implementation of the different steps.

4.1 Setting of the Case Study

Our "Omni-Channel-Banking" case-study supports a variety of different channels to access banking services. Figure 6 shows its overall architecture.

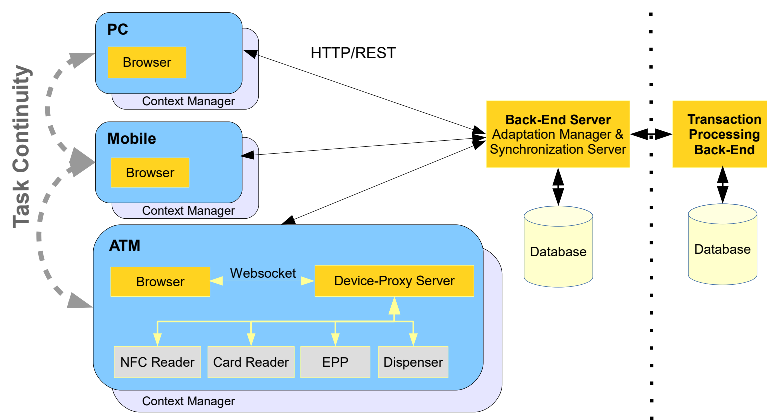

Fig. 6. Case-Study application architecture

On each device - *PC*, *Mobile*, *ATM* - the client application is running as a single-page web application inside a browser. Each device has an underlying *Context Manager* that observes and delivers context information to the *Back-End Server*, which is responsible for

- serving an application to the browser, adapted to a particular context of use,
- serving application specific data to the client via HTTP/REST,
- managing application state and UI adaptation,
- requesting information from a *Transaction Processing Back-End* and serving it to the client,
- sending financial transactions to the *Transaction Processing Back-End* for execution.

The data format for all data exchanged through HTTP/REST requests is *JavaScript Object Notation* (JSON).

The *Transaction Processing Back-End* is not part of our application, but represents an existing infrastructure for processing financial transactions. The *Back-End Server* communicates with this transaction processing system. The communication protocol between the *Transaction Processing Back-End* and our sample application's *Back-End Server* depends on an existing infrastructure. Thus, the *Back-End Server* needs to provide a custom adapter for interfacing with this system.

In our case study, PC and mobile applications are identical concerning their functionality. The main difference comes from adaptation to different context of use scenarios (e.g. screen sizes, operation through keypad/touch screen etc.). This also includes spreading of functiontionality on the mobile device over multiple dialogs, compared to the PC application. In contrast to PC and mobile clients, the application architecture of the ATM client is significantly different. This is due to the need for supporting a whole variety of ATM specific hardware devices, like *NFC Reader*, *Card Reader*, *Encrypting Pin Pad (EPP)*, *Cash Dispenser*, etc. For interoperability reasons, ATM vendors are using a common software stack called XFS, which is layered on top of device specific drivers. XFS stands for *Extensions for Financial Services* and is standardized by CEN, the *European Commitee for Standardization*. Since a browser itself can not directly access the XFS-API, we delegate device control to a *Device-Proxy Server* running directly on the ATM.

4.2 UI/Adaptation Modeling and Transformation

For realizing the modeling (UI and adaptation) and transformation step of the engineering approach, we have implemented a model-based UI development (MBUID) process which is depicted in figure 7. This MBUID process supports the modeling of UIs and adaptation rules as well as their transformation to final user interfaces, which are the view parts of the single-page application rendered as HTML5 by the browser. Based on the open source IFML Editor Eclipse plugin developers are able to specify the domain and abstract UI model. In figure

8 the domain model and an excerpt of the abstract UI model showing the login view are depicted for the example scenario of our case study. As a complementary modeling step, the IFML extensions *Context-IFML* and *Adapt-IFML* allow the specification of adaptation rules for UI adaptation at runtime. For showing the structure of such adaptation rules, figure 9 shows two exemplary adaptation rules represented as a table. For transforming these models into final web UI views, we implemented an Xtend plugin that maps the IFML model elements to specific HTML5 elements. The Xtend plugin includes different Xtend templates to transfer the IFML source model into web UIs supporting different context of use parameters.

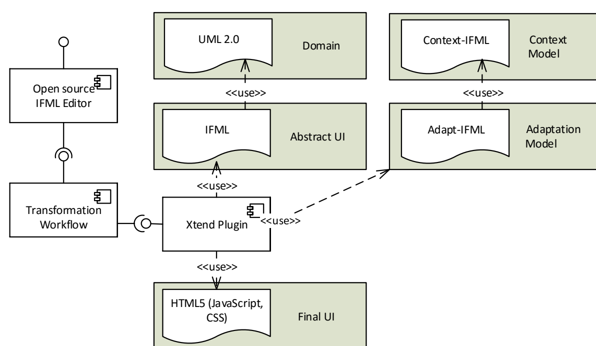

Fig. 7. Implemented model-based UI development process

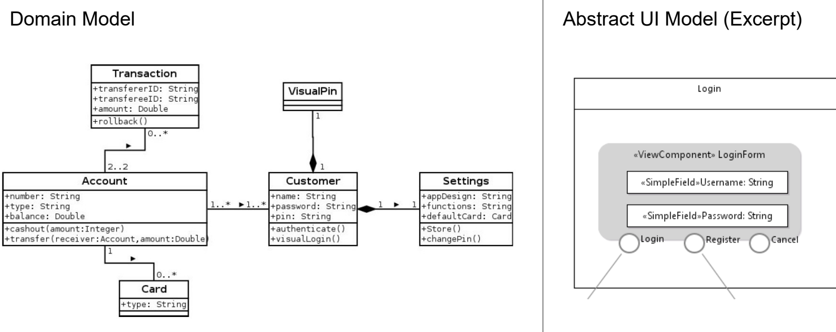

Fig. 8. Exemplary Domain Model and excerpt of the Abstract UI Model

Adapation Rule - Example 1	
Event	User.Preferences.FontSize = 18
Condition	Value.DefaultFontSize < 18 ?
Action	Layout_ChangeOperation.ChangeFontSize()

Adapation Rule - Example 2	
Event	Platform.Device = Mobile Platform.Device.Screen = small
Condition	Operator.Compare(DefaulScreenSize, TargetScreenSize)
Action	Layout_ChangeOperation.SplitViewContainer()

Fig. 9. Examples for Adaptation Rules

During the transformation process, the application's view is built upon basic components with a custom look & feel, like buttons, text input fields, dropdown lists, tables, etc. As a basis for these components, we did not use AngularJS directives, but implemented components based on the *HTML5 Web Components*[4] specification promoted by Google as W3C standard.

Our custom components are sensitive to the context of use they are being used in and adapt themselves accordingly. On mobile devices and on the ATM, for example, buttons are larger and more suitable for touch operation than on desktop devices.

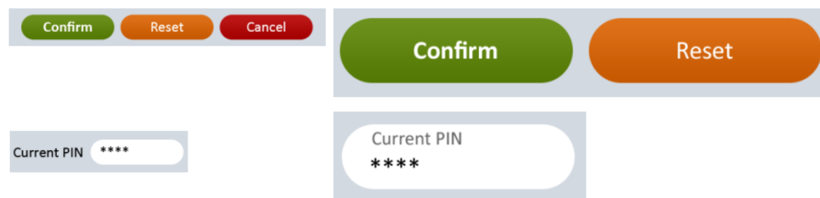

Fig. 10. Buttons and text fields for Desktop and Mobile

Figure 10 shows buttons and text input fields. Their desktop representation is depicted on the left side, their mobile appearance on the right side of the picture.

During the transformation process for all device classes, a button is created the same way:

```
<komos-button colorscheme="cs1" ng-click="confirm()">
  Confirm
<komos-button>
```

The following example shows how to create a text input field with a label by mapping an IFML simple field element to the following code snippet:

```
<komos-textfield label="Current PIN" ng-model="model.currentPin">
  </komos-textfield>
```

In order to provide a unified layout management for our application, our model-to-text (M2T) transformation process implements a custom layout man-

[4] https://www.w3.org/TR/components-intro

ager. It provides an easy to use grid layout system, based on row and column elements realized as AngularJS directives. Under the hood, it uses HTML5 Flexbox. Figure 11 shows the generated code snippet to create the login dialog that is depicted below.

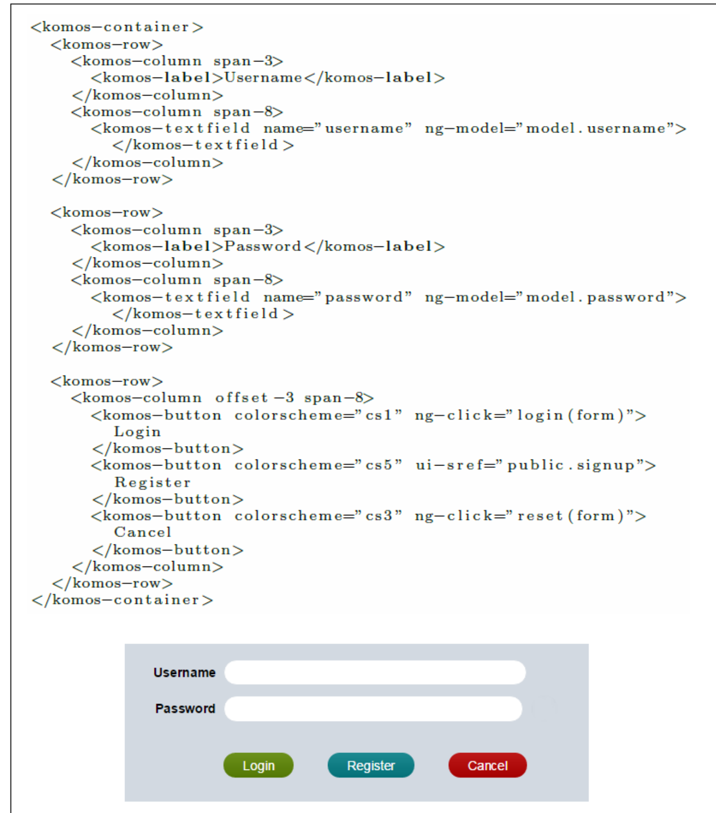

Fig. 11. Desktop UI: generated code snippet and corresponding login dialog

Beside the mapping from abstract UI models to HTML5 views, we have established a transformation process for translating the adaptation rules into the target language of the final user interface, so that runtime adapation of the UI can be supported. As we have decided to use the Nools[5] engine as a rule-based execution environment for executing context-adaptive UIs (see subsection 4.4 for details), we implemented particular Xtend templates to transform the *Adapt-IFML* adaptation rules to rules specified with the Nools DSL. The result of such a generated adaptation rule is shown in figure 12.

[5] https://github.com/C2FO/nools

```
rule AtmProfile {
    when {
        p : Platform p.type == "atm";
    }
    then {
        displayProperties.type = p.type;
        displayProperties.controlActionType = "atm-action-button";
        displayProperties.singleChoiceType = "single-choice-list";
    }
}
```

Fig. 12. Example of a generated adaptation rule that is represented in the Nools DSL

4.3 Execution and Adaptation (Front-End)

While the previous subsection presented our MBUID process to support the modeling and generation of view aspects of the *Front-End*, this subsection deals with the execution of the resulting UIs. In this context, we especially present the controller part of the *Front-End*, which is responsible for application logic and communication with the *Back-End Server*. In conjunction with this topic, we also present the aspect of channel handover and task-continuity. The aspect of UI adaptation at runtime will be explained in the next subsection.

As shown in figure 13, the *Front-End* consists of a HTML5/JavaScript single-page application running in a web browser. It exchanges JSON messages with the *Back-End Server* through HTTP/REST.

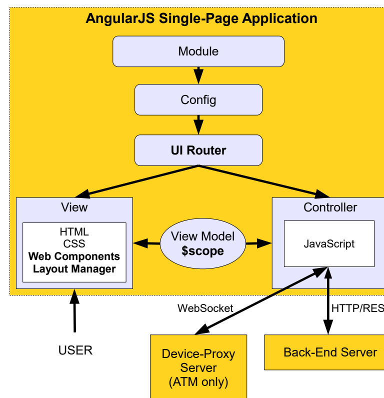

Fig. 13. Front-End Architecture

The browser application's main building blocks are:

– AngularJS[6]: Google's open-source web application framework for developing single-page applications in JavaScript

[6] https://angularjs.org

- UI Router[7]: flexible client-side routing with nested views in AngularJS
- Web Components: UI components with custom look & feel
- Layout Manager: custom layout manager

AngularJS supports the model-view-controller (MVC) design pattern by decoupling the application's presentation layer, which is defined through HTML5 (see previous subsection), from the model and application logic by two-way databinding through a `$scope` object. In addition, AngularJS provides a variety of other services, including modularization and definition of custom directives.

UI Router is the client-side routing component of AngularJS and the central key component to implement task continuity. The developer assigns a particular application state, identified by a name (`protected.main`), with a view (`main.html`) and a controller (`MainCtrl`):

```
angular.module('komosApp').config(function ($stateProvider) {
  $stateProvider
    .state('protected.main', {
      url: '/',
      templateUrl: 'protected/main/main.html',
      controller: 'MainCtrl',
      authenticate: true
  });
});
```

In order to support task continuity and transfer application state between devices, the current state name and its associated context are saved to the *Back-End Server*.

Inside a view controller and prior to saving a state, all context information necessary for recovery is added to a state-context object. This includes the UI's view-model, as well as any other necessary information associated with the current state.

```
var context = {
  // the view model:
  viewModel: $scope.model,
  // state specific arbitrary properties:
  param1: someValue,
  data: someData
};

PersistStateService.save('protected.main', context, function (err, data) {
  if (err) model.errors.message = err.data.message;
});
```

We implemented an AngularJS service named `PersistStateService`, which converts the object `context` to JSON and sends it to the *Back-End Server*, where it is stored under the name of the state, e.g. `protected.main`. To invoke a previously saved state, the application just needs to retrieve the current state name and invoke it:

```
$rootScope.$state.go('protected.main');
```

On instantiation of the AngularJS controller associated with this state, the controller calls the service's restore method to retrieve the previously stored information:

[7] https://github.com/angular-ui/ui-router/wiki

```
PersistStateService.restore('protected.main', function (err, context) {
  if (err) {
    model.errors.message = err.data.message;
  } else {
    // context now contains the previously saved information
    $scope.model = context.viewModel; // this updates the UI!!
    someValue = context.param1;
    someData = context.data;
  }
});
```

Both, saving and retrieving context data for a state happens within the same controller. Each controller knows exactly which data needs to be saved in order to be able to restore itself. This information is hidden from other parts of the application. The only knowledge necessary from the outside is the name of the state, `protected.main` in our example.

Because of AngularJS' two-way data-binding, assigning the view-model to `$scope.model` immediately updates the view.

4.4 Execution and Adaptation (Back-End)

The application's *Back-End* is implemented in JavaScript (see figure 14) and uses Node.js[8] as its runtime environment. It is built upon Google's V8 JavaScript engine also used by Google Chrome and provides a high-performance runtime environment for non-blocking and event-driven programming.

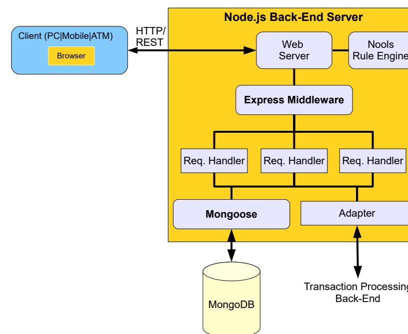

Fig. 14. Back-End Architecture

ExpressJS[9], which is a middleware for Node.js, provides components for processing of requests and routing. An application sets up request handlers, which are automatically invoked when a client request arrives. Within a request handler, the request is processed, a response is prepared and returned. Request handlers communicate with the database or *Transaction Processing Back-End*.

[8] https://nodejs.org
[9] https://expressjs.com

Fig. 15. Context-Adaptive UIs for a Cross-Channel Banking Web Application supporting Task-Continuity

For execution and adaptation of the final UIs at runtime, our Back-End Server integrates *Nools* as a rule-based execution environment for representing the *Adaptation Manager*. *Nools* is an efficient RETE-based rule engine for Node.js written in JavaScript and provides an API and rule language (DSL) for specifying fact and rules. By transforming context information to facts and translating the specified adaptation rules to *Nools* rules, the *Nools* engine supports the automatic adaptation of UIs at runtime as reaction to dynamically changing context of use parameters.

The document database *MongoDB*[10] belongs into the category of NoSQL ("Not Only SQL") databases. In this context, a "document" consists of a user-defined data structure of key-value pairs, which is associated with a key. Documents can also contain other documents. The schema of a database is dynamic and can be modified at runtime. To access the database in an object-oriented fashion, we use an *Object Document Mapper* called *Mongoose* on top of MongoDB's Node.js driver.

The instantiation of our engineering approach and interaction of all described technologies resulted in the demonstrator which is shown in figure 15. Our demonstrator shows the implementation of context-adaptive UIs for a cross-channel banking web-application that supports different channels (Desktop, Tablet, and ATM) for a cash payout process enabling task-continuity for the customers.

5 Conclusion and Outlook

This paper presents a model-based engineering approach that supports systematic and efficient development of context-adaptive UIs for heterogeneous and dynamically changing contexts of use (user, platform, environment). The proposed engineering approach supports modeling, transformation and execution of context-adaptive UIs that enable a personalized, flexible and task-continuous usage of cross-channel applications. The feasibility of the approach was shown

[10] https://www.mongodb.org

based on an industrial case study, where the implementation of context-adaptive UIs for a cross-channel banking web application is presented. Furthermore, industrial experiences resulted in positive feedback regarding the applicability and efficiency of the approach. However, we plan to evaluate the efficiency and effectiveness of the approach in more complex development scenarios. Future work will also cover the evaluation of usability aspects for end-users of the generated and adapted UIs as well as for the developers using the tools and languages proposed in our approach.

References

1. Roland Petrasch. 2007. Model Based User Interface Design: Model Driven Architecture und HCI Patterns. In: GI Software-technik-Trends, Band 27, Heft 3, 5-10.
2. Fabio Paternò and Carmen Santoro. 2012. A logical framework for multi-device user interfaces. In Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems (EICS '12). ACM, New York, NY, USA, 45-50.
3. Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput.-Hum. Interact.
4. Enes Yigitbas, Holger Fischer, Thomas Kern, and Volker Paelke. 2014. Model-Based Development of Adaptive UIs for Multi-channel Self-service Systems. In Proceedings of the International Conference on Human-Centered Software Engineering (HCSE '14). LNCS, Springer, 267-274.
5. James Lin and James A. Landay. 2008. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08). ACM, New York, NY, USA, 1313-1322.
6. Jan Meskens, Jo Vermeulen, Kris Luyten, and Karin Coninx. 2008. Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In Proceedings of the working conference on Advanced visual interfaces (AVI '08). ACM, New York, NY, USA, 233-240.
7. Fabio Paternò, Carmen Santoro, and Antonio Scorcia. 2010. Ambient Intelligence for Supporting Task Continuity across Multiple Devices and Implementation Languages. Comput. J. 53, 8 (October 2010), 1210-1228.
8. Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. 2012. Push and pull of web user interfaces in multi-device environments. In Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI '12). ACM, New York, NY, USA, 10-17.
9. Enes Yigitbas, Stefan Sauer, and Gregor Engels. 2015. A Model-Based Framework for Multi-Adaptive Migratory User Interfaces. In Proceedings of the 17th International Conference on Human-Computer Interaction. Springer, LNCS, vol. 9170, pp. 563-572.
10. Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. 2014. Interactive development of cross-device user interfaces. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14).
11. Jishuo Yang and Daniel Wigdor. 2014. Panelrama: enabling easy specification of cross-device web applications. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 2783-2792.

12. Peter Hamilton and Daniel J. Wigdor. 2014. Conductor: enabling and understanding cross-device interaction. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 2773-2782.
13. J. Coutaz. 2010. User Interface Plasticity: Model Driven Engineering to the Limit! In Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems. ACM, 18.
14. Anthony F. Norcio and Jaki Stanley. 1989. Adaptive Human-Computer Interfaces: A Literature Survey and Perspective. In IEEE Transactions on Systems, Man, and Cybernetics, 19, 399-408.
15. Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2014. Adaptive Model-Driven User Interface Development Systems. ACM Comput. Surv. 47, 1, Article 9 (May 2014), 33 pages.
16. Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. 2014. Integrating adaptive user interface capabilities in enterprise applications. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 712-723.
17. Peter Brusilovsky. 2001. Adaptive Hypermedia. User Modeling and User-Adapted Interaction 11, 1-2 (March 2001), 87-110.
18. Reinhard Oppermann. 1989. Individualisierte Systemnutzung. In GI - 19. Jahrestagung, I, Computergesttzter Arbeitsplatz, Manfred Paul (Ed.). Springer-Verlag, London, UK, UK, 131-145.
19. Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically generating personalized user interfaces with Supple. Artif. Intell. 174, 12-13 (August 2010), 910-950.
20. Mladjan Jovanovic, Dusan Starcevic, and Zoran Jovanovic. 2014. Bridging User Context and Design Models to Build Adaptive User Interfaces. In Proceedings of the International Conference on Human-Centered Software Engineering (HCSE '14). LNCS, Springer, 36-56.
21. Stefan Sauer. 2011. Applying Meta-Modeling for the Definition of Model-Driven Development Methods of Advanced User Interfaces. In Model-Driven Development of Advanced User Interfaces. Springer, 67-86.
22. Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. Computer 36, 1 (January 2003), 41-50.
23. Tim Clerckx, Kris Luyten, and Karin Coninx. 2004. DynaMo-AID: A Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development. In Engineering Human Computer Interaction and Interactive Systems. LNCS, Springer, 77-95.
24. Kris Luyten, Jan Van den Bergh, Chris Vandervelpen, and Karin Coninx. 2006. Designing distributed user interfaces for ambient intelligent environments using models and simulations. In Computers and Graphics. Pergamon, 702-713.
25. Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Vctor Lpez-Jaquero. 2004. USIXML: A Language Supporting Multi-path Development of User Interfaces. In Engineering Human Computer Interaction and Interactive Systems. LNCS, Springer, 200-220.
26. Alexandre Demeure, Galle Calvary, and Karin Coninx. 2008. COMET(s), A Software Architecture Style and an Interactors Toolkit for Plastic User Interfaces. In DSV-IS 2008. LNCS, Springer, 225-237.
27. Giuseppe Ghiani, Marco Manca, Fabio Patern, and Claudio Porta. 2014. Beyond Responsive Design: Context-Dependent Multimodal Augmentation of Web Applications. In MobiWIS 2014. LNCS, Springer, 71-85.