Model-Driven Engineering of Self-Adaptive User Interfaces

PADERBORN UNIVERSITY
The University for the Information Society

Enes Yiğitbaş

Faculty of Computer Science, Electrical Engineering and Mathematics
Paderborn University

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Paderborn, December 2019

I would like to dedicate this thesis to my loving parents . . .

Acknowledgements

Writing this thesis would not have been possible without the support of numerous people. Hence, I would like to thank the following people for sharing their experience, time, and patience.

First of all, I would like to thank my advisor and mentor, Prof. Dr. Gregor Engels, for his efforts to drive me to success with wise advice, patient guidance, and kind encouragement. Gregor, thanks for your trust, all the opportunities and the fruitful environment for learning and growing into many different directions. Talking about opportunities, I further thank Dr. Stefan Sauer who allowed me to work in inspiring projects at the s-lab – Software Quality Lab and si-lab – Software Innovation Lab. I would also like to thank my committee members Prof. Dr. Philippe Palanque, Prof. Dr. Regina Bernhaupt, and Jun. Prof. Dr. Anthony Anjorin for the friendly guidance and thought-provoking suggestions while carefully reviewing my work to strengthen and elevate its quality.

I thank all my current and past colleagues from the Database and Information Systems research group for providing such an excellent, professional, and friendly atmosphere in our research group.

Abstract

The user interface (UI) is a key component of any interactive software application and is crucial for the acceptance of the application as a whole. Modern UIs are increasingly expected to be plastic, in the sense that they retain a constant level of usability, even when subjected to context (user, platform, and environment) changes at runtime. Self-adaptive User Interfaces (SAUIs) have been promoted as a solution for context variability due to their ability to automatically detect context changes and adapt to the current context-of-use at runtime.

However, engineering SAUIs is a challenging and complex task. Concerning development, aspects such as context management and UI adaptation further increase complexity compared to the development of classical UIs. Thus, an integrated development approach enabling context management and UI adaptation is required. Concerning evaluation, usability plays a crucial role for acceptance of SAUIs. Especially the usability aspect end-user satisfaction regarding UI adaptations at runtime is important to assess whether end-users accept the quality of use. As the UI and the context-of-use are both constantly changing, usability evaluation becomes more complex.

To address the mentioned issues, we introduce a model-driven engineering approach for SAUIs with twofold contribution:

On the one hand, we introduce a model-driven development approach for SAUIs. Our development approach supports modeling, transformation and execution of SAUIs. The development approach covers an integrated model-driven development solution where a classical model-driven development of UIs is coupled with a model-driven development of context-of-use and UI adaptation rules. We base our approach on the core UI modeling language Interaction Flow Modeling Language (*IFML*) and introduce new modeling languages for context-of-use (*ContextML*) and UI adaptation rules (*AdaptML*). The generated UI code, based on the *IFML* model, is coupled with the *Context* and *Adaptation Services*, generated from the *ContextML* context model and *AdaptML* adaptation model, respectively. The integration of the generated artifacts, namely UI code, *Context*, and *Adaptation Services* in an overall rule-based execution environment, enables runtime UI adaptation.

On the other hand, we present a novel on-the-fly usability testing solution for SAUIs. It allows to evaluate the end-user satisfaction of SAUIs by combining context monitoring together with collection of instant user feedback. Our model-driven development approach serves as a basis for driving the on-the-fly usability test. Based on the underlying models (e.g., adaptation models), we can derive usability interview questions for assessing the acceptance of various UI adaptation features. The developed usability evaluation solution enables us to continuously track various context information data, collect user feedback, as well as perform a data-driven usability evaluation.

The overall evaluation of our model-driven engineering approach focuses on the applicability of our model-driven development approach as well as usability evaluation of the resulting self-adaptive UIs. The applicability of our model-driven development approach is demonstrated by two case-studies showing the development of self-adaptive UIs for a library application and an e-mail application. Furthermore, the results of a usability study based on the derived e-mail application are presented.

Zusammenfassung

Die Benutzungsschnittstelle (engl. User Interface, UI) ist eine Schlüsselkomponente jeder interaktiven Softwareanwendung und von entscheidender Bedeutung für die Akzeptanz der Gesamtanwendung. Es wird zunehmend erwartet, dass moderne UIs in dem Sinne plastisch sind, dass sie ein konstantes Maß an Benutzerfreundlichkeit behalten, selbst wenn sie zur Laufzeit Änderungen des Nutzungskontexts (Benutzer, Plattform und Umgebung) unterliegen. Selbst-adaptive UIs (SAUIs) wurden als Lösung für Kontextvariabilität eingeführt, da sie automatisch Kontextänderungen erkennen und sich zur Laufzeit an den aktuellen Nutzungskontext anpassen können.

Das Engineering von SAUIs ist jedoch eine herausfordernde und komplexe Aufgabe. In Bezug auf die Entwicklung erhöhen Aspekte wie das Kontextmanagement und die Anpassung des UIs die Komplexität im Vergleich zur Entwicklung klassischer UIs. Daher ist ein integrierter Entwicklungsansatz erforderlich, der das Kontextmanagement und die Anpassung des UIs ermöglicht. In Bezug auf die Usability Evaluation spielt die Benutzerfreundlichkeit eine entscheidende Rolle für die Akzeptanz von SAUIs. Insbesondere ist der Usability-Aspekt der Nutzerzufriedenheit im Hinblick auf UI-Anpassungen zur Laufzeit wichtig, um zu beurteilen, ob Nutzer die Nutzungsqualität akzeptieren. Da sich sowohl das UI als auch der Nutzungskontext ständig ändern, wird die Bewertung der Benutzerfreundlichkeit komplexer.

Um die genannten Probleme anzugehen, führen wir einen modellgetriebenen Engineering-Ansatz für SAUIs mit zwei Beiträgen ein:

Einerseits führen wir einen modellgetriebenen Entwicklungsansatz für SAUIs ein. Unser Entwicklungsansatz unterstützt die Modellierung, Transformation und Ausführung von SAUIs. Der Entwicklungsansatz umfasst eine integrierte modellgetriebene Entwicklungslösung, bei der eine klassische modellgetriebene Entwicklung von UIs mit einer modellgetriebenen Entwicklung von Nutzungskontext- und UI-Anpassungsregeln gekoppelt ist. Wir stützen unseren Ansatz auf die zentrale UI-Modellierungssprache Interaction Flow Modeling Language (*IFML*) und führen neue Modellierungssprachen für Nutzungskontext (ContextML) und UI Anpassungsregeln (AdaptML) ein. Der generierte UI-Code, der auf dem IFML-Modell basiert, wird mit den Kontext- und Anpassungsdiensten gekoppelt, die aus dem ContextML-Kontextmodell bzw. dem AdaptML-Anpassungsmodell generiert werden. Die

Integration der generierten Artefakte, nämlich UI-Code, Kontext- und Anpassungsdienste, in eine allgemeine regelbasierte Ausführungsumgebung ermöglicht die Anpassung des UI zur Laufzeit.

Zum anderen präsentieren wir eine neuartigen Ansatz zum Testen der Benutzerfreundlichkeit von SAUIs, der als "‘On-the-Fly-Usability-Test"’ bezeichnet wird. Dieser ermöglicht die Bewertung der Nutzerzufriedenheit, indem die Kontextüberwachung mit der Erfassung von sofortigem Benutzerfeedback kombiniert wird. Unser modellgetriebener Entwicklungsansatz dient als Grundlage für den erwähnten "‘On-the-Fly-Usability-Test"’. Basierend auf den zugrunde liegenden Modellen (z.B. Anpassungsmodellen) können wir Usability-Interviewfragen ableiten, um die Akzeptanz verschiedener UI-Anpassungsmerkmale zu bewerten. Die entwickelte Usability-Evaluierungslösung ermöglicht es uns, verschiedene Kontextinformationsdaten kontinuierlich zu verfolgen, Benutzerrückmeldungen zu sammeln sowie eine datengesteuerte Usability-Evaluierung durchzuführen.

Die Gesamtbewertung unseres modellgetriebenen Engineering-Ansatzes konzentriert sich auf die Anwendbarkeit unseres modellgetriebenen Entwicklungsansatzes sowie auf die Usability Evaluation der resultierenden SAUIs. Die Anwendbarkeit unseres modellgetriebenen Entwicklungsansatzes wird anhand von zwei Fallstudien demonstriert, die die Entwicklung von SAUIs für eine Bibliotheksanwendung und eine E-Mail-Anwendung zeigen. Darüber hinaus werden die Ergebnisse einer Usability-Studie basierend auf der abgeleiteten E-Mail-Anwendung vorgestellt.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In this chapter, we describe the motivation and problem statement for this thesis. We point out our solution approach and main contributions. We present an overview of publications published in the context of this thesis and provide a structural overview of the thesis.

1.1 Motivation

The user interface (UI) is a key component of any interactive software application and is crucial for the acceptance of the application as a whole. Acceptance and usability of a user interface are highly influenced by its context-of-use, which is defined in terms of the user, platform, and environment [CCT$^+$03].

With the advent of the Internet of Things (IoT) and mobile applications, users are nowadays surrounded by a broad range of networked interaction devices (e.g., smartphones, smartwatches, tablets, terminals, etc.) for carrying out their work and 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 working environment) have to be supported, it is no longer sufficient to provide a single "one-size-fits-all" user interface. Still, most of today's user interfaces are typically designed with the assumption that they are going to be used by an able-bodied user, who is located in a stable environment, and who is using dedicated platform devices [Mot13]. However, this assumption leads to a significant gap between the actual needs of end users and what technology offers them, because in reality users comprise a heterogeneous group with different abilities, preferences etc., that interact via different devices in distinct working environments. The problem increases even more if we consider dynamic changes in the context-of-use parameters at runtime, e.g., changes of a specific user role, the device type, or environmental factors such as brightness or loudness level when the interaction is taking place. In this case, the "one-size-fits-all" approach is unable to accommodate all the cases of variability in the context-of-use, in many cases leading to a diminished user experience [ABY14a].

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 [ABY14a], for example by changing the UI's navigational flow, its content, or layout. A key goal behind adaptive UIs is plasticity characterizing a UI's ability to preserve its usability across multiple contexts-of-use [Cou10]. A very important prerequisite for UI adaptivity is context-awareness. If the UI is aware of its context and is able to detect context changes, then it can trigger adaptations in response to those changes in order to preserve its usability. UIs that are able to automatically capture context information using sensors from heterogeneous sources, and monitor dynamic context changes to enable UI adaptation at runtime, are called *self-adaptive UIs*. Summing up, from the end users' perspective, a user interface should be flexible and self-adaptive in order to automatically monitor context changes, and adjust itself according to the changing context-of-use parameters to provide a high usability.

Changing the view from the users' demands for an interactive system's UI to that of the developers' who implement such flexible interactive systems, one major challenge can be identified: complexity in the development process of such flexible user interfaces. The development of user interfaces for interactive software systems is a time-consuming and error-prone task [MR92]. By analyzing a number of different software applications, it was found that about 48% of the source code, about 45% of the total development time, about 50% of the implementation time, and about 37% of the maintenance time is required for aspects regarding user interfaces. A more recent study [Pet07] argues that the time effort needed for implementing user interfaces is still at least 50% and justifies it with the increasing spread of interactive systems and their requirements due to the progressive growth in the amount and diversity of technological devices available on the market.

Relating above mentioned facts to the realization of self-adaptive UIs, shows that additional complexity arises to cope with dynamically changing context-of-use parameters and automatically adapting the UI at runtime. Building multiple UIs for the same functionality due to context variability is a difficult task since context changes can lead to a combinatorial explosion of the number of possible adaptations and there is a high cost incurred by manually developing multiple versions of the UI [ABY14a]. Hence, the development of advanced UIs, such as self-adaptive UIs, demands for sophisticated engineering processes and methods.

Not only the development, but also the evaluation of self-adaptive UIs is a challenging task that should be addressed by adequate methods and techniques. Concerning evaluation, usability plays a crucial role for the acceptance of self-adaptive UIs. Especially the usability aspect end-user satisfaction, regarding UI adaptations at runtime, is important to assess whether end-users accept the quality of use. As the UI and the context-of-use are both constantly changing, usability evaluation becomes more complex. Furthermore, classical usability evaluation methods such as usability tests, interviews or cognitive walkthroughs mostly focus on a posteriori analysis techniques and do not fully consider the current context-of-use at runtime.

To address the mentioned issues regarding development and evaluation of self-adaptive UIs, we introduce a model-driven engineering approach for self-adaptive UIs. On the one hand, it consists of a model-driven development approach for self-adaptive UIs. It supports developers in modeling, transformation, and execution of self-adaptive UIs. On the other hand, the engineering approach introduces a novel on-the-fly usability testing solution for self-adaptive UIs. It allows to evaluate the end-user satisfaction of self-adaptive UIs by combining context monitoring together with collection of instant user feedback. The developed usability evaluation solution enables us to continuously track various context information data and user feedback as well as to perform a data-driven usability evaluation.

1.2 Problem Statement

Among the various components of software applications, the user interface is especially important as it connects the end-users to the actual functionality of an interactive system. As stated before, the effort of implementing an application's user interface constitutes at least 50% of the total implementation effort [Pet07]. 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 before.

Providing an adequate user interface that is suitable for the current contextual situation of the end-user is a complex task that imposes special challenges beyond classical UI development that targets a fixed set of users or platforms.

The first challenge is to automatically capture context information about the user, platform, and environment by making use of different sensors. The main goal is to continuously monitor the contextual parameters and to detect context changes. The second challenge is to analyze the collected context information data and to argue based on those data whether UI adaptation is needed to react on the context changes to provide a more suitable user interface. Finally, if some UI adaptation operations are needed to overcome the situational mismatch between shown UI and the current context-of-use, the execution of the UI adaptation operations is used to reflect the changes on the UI. These adaptation operations could be for example layout changes, content changes or changes in the navigational flow of the UI.

Tackling the above described challenges of context management and UI adaptation is already a complex task on its own. Moreover, additional complexity is given due to the fact that today's user interfaces of interactive systems become increasingly complex due to new interaction paradigms, use of innovative technologies, multimedia, and interaction modalities. Their development thus demands for sophisticated processes and methods, as they are deployed in software engineering [Sau11].

One promising way for dealing with the complex development task of self-adaptive UIs is model-based or model-driven UI development. Model-Based UI Development (MBUID) provides a means to decrease the development effort through the use of high-level models that are refined for a certain context-of-use (e.g. user, platform, or environment). Model-driven UI development (MDUID), in addition, puts the models at the center of the refinement process and applies automated model-to-model and model-to-code transformations to transform high-level UI models to source code of the final UI.

In the past, various MDUID approaches were proposed to support the efficient development of UIs. Widely studied approaches are based on UsiXML [LVM$^+$04], MARIA [PSS09], and IFML [BF14] that support the abstract modeling of user interfaces and their transformation to final user interfaces. However, in classical MDUID approaches, specific aspects and requirements regarding the development of self-adaptive UIs such as context management and UI adaptation are not covered in an integrated manner. Those aspects introduce additional complexity and need to be taken into account. As most of the existing approaches focus on special aspects of UI adaptation, an integrated model-driven development approach for self-adaptive UIs is not fully covered.

In practice, especially in the context of web design and web technologies, the paradigm of Responsive Web Design (RWD)[1] is widely used to adapt the layout of a web page in response to the characteristics of the used device. While RWD adaptations are mainly focusing on the contextual parameter *Platform*, considering device characteristics such as screen size or resolution, our scope in this thesis is going far beyond by focusing on a holistic dimension of contextual parameters where various context-of-use situations can be handled. Beside RWD and MDUID approaches in general, there are existing specific approaches such as Supple [GWW10], MASP [FBA06], MyUI [PHJS12] or RBUIS [ABY16] which present methods, techniques and tools for the development of adaptive UIs. Although all of these approaches varying in the used technique (rule-based, optimization-based, learning-based) show a successful implementation of UI adaptation features and their practical usage in different domains, an integrated model-driven engineering approach for self-adaptive UIs is not fully addressed. Such an integrated engineering approach should support the development and usability evaluation of self-adaptive UIs, which leads to the main research question of this thesis:

(RQ) *How can we support the model-driven engineering of self-adaptive UIs for heterogeneous contexts-of-use?*

Regarding the development of self-adaptive UIs, an integrated model-driven development approach is required that should provide dedicated languages for context and adaptation modeling which complement UI modeling. Following the idea of MDUID and reducing the complexity in the development of self-adaptive UIs, specific code generators are further required. Beside the *Final UI* (FUI) code, these are code for *Context Services* to monitor the context information at runtime and also code for *Adaptation Services* to represent adaptation logic and enable UI adaptation at runtime. Thus, for addressing the development part of our model-driven engineering approach for self-adaptive UIs, the following challenges were identified:

Context Management Challenges:

- *C1: Specification of contextual parameters*: A context modeling language is required for specifying different contexts-of-use. Such a context modeling language should

[1]https://www.w3schools.com/whatis/whatis_responsive.asp

enable modeling of different contextual situations that can occur during usage of the UI. With the help of this language, developers should be able to specify needed context sensor services to monitor various contextual parameters.

- *C2: Generation of Context Services*: A *Context Service* provides context information (e.g., brightness level, mood of the user etc.) which are specified in the context model by accessing hardware sensors. A transformation method for automatic generation of heterogeneous *Context Services* based on the context model is needed. The specified context model serves as input for generating code for the required *Context Services* which enable the monitoring of context information and triggering the adaptation at runtime.
- *C3: Execution of Context Services and runtime monitoring*: An execution environment is required for executing the generated *Context Services*. For supporting runtime monitoring of dynamic context changes, the generated *Context Services* should observe the context sensors and provide context information data within the overall UI execution environment.

UI Adaptation Challenges:

- *C4: Specification of UI adaptation rules*: An adaptation modeling language is required for specifying UI adaptation changes in an abstract manner. With the help of this language, UI designers should be able to specify various UI adaptation rules for different contexts-of-use which can adapt the UI at runtime.
- *C5: Generation of UI Adaptation Services*: An *Adaptation Service* is responsible for monitoring the context information provided by the context service and adapting the UI at runtime. A transformation method for automatic generation of different *Adaptation Services* is needed. Based on the specified abstract UI adaptation rules, the code for the executable *Adaptation Services* needs to be generated for supporting UI adaptation capabilities at runtime.
- *C6: Execution of UI adaptation at runtime*: An integrated execution environment is required for executing the generated *Context* and *Adaptation Services*. For supporting runtime UI adaptation enabling automatic reaction to dynamic context-of-use changes, the generated *Adaptation Services* need to be coupled with generated code for the UI and *Context Services* as well as integrated in an overall UI execution environment.

While above described challenges are related to development aspects, the aspect of usability evaluation should be also integrated in the model-driven engineering approach for self-adaptive UIs. With this regard, the usability aspect *end-user satisfaction* regarding UI adaptations at runtime is important for acceptance of self-adaptive UIs. Focusing the ubiquitous domain of mobile UI platforms, where dynamically changing context-of-use situations are usual, usability evaluation of UI adaptations is still a challenging task. One can possibly use classical usability evaluation methods such as usability tests, interviews or cognitive walkthroughs. However, these methods are not sufficient for a proper evaluation of UI adaptation features. The reason is that classical usability evaluation methods mostly focus on *a posteriori* analysis techniques. However, usability testing of UI adaptation features should consider the current context-of-use when the adaptation is triggered and also the user's feedback should be taken into account. Therefore, a solution for usability testing of UI adaptations should be incorporated in the model-driven engineering approach for self-adaptive UIs, which tackles the following challenges.

Usability Evaluation Challenges:

(C7) *On-the-fly Testing*: The usability evaluation method enables testing the acceptance of UI adaptation features at runtime despite the dynamic nature of adaptive UIs where context parameter and the UI itself are continuously changing.

(C8) *Collection of Instant User Feedback and Context-of-use Data*: The usability evaluation method enables testing the acceptance of UI adaptation features at the very moment when the adaptations occur incorporating the current context-of-use and instant user feedback.

1.3 Solution Overview and Scientific Contributions

To address the above mentioned challenges, the main goal of this thesis is to support model-driven engineering of self-adaptive UIs. As depicted in Figure 1.1, the contribution of this thesis is twofold. On the one hand, we present a model-driven development approach for self-adaptive UIs which covers relevant aspects such as UI, context, and adaptation. On the other hand, we introduce a novel data-driven usability evaluation approach which allows on-the-fly usability testing of self-adaptive UIs and supports a context- and data-driven satisfaction analysis of self-adaptive UIs. In this constellation, the model-driven development approach with its underlying models (e.g. adaptation models) serves as a basis for the on-the-fly usability testing solution as we can derive usability test questions from the specified UI adaptation rules. In the following, we will describe the engineering approach in more detail.

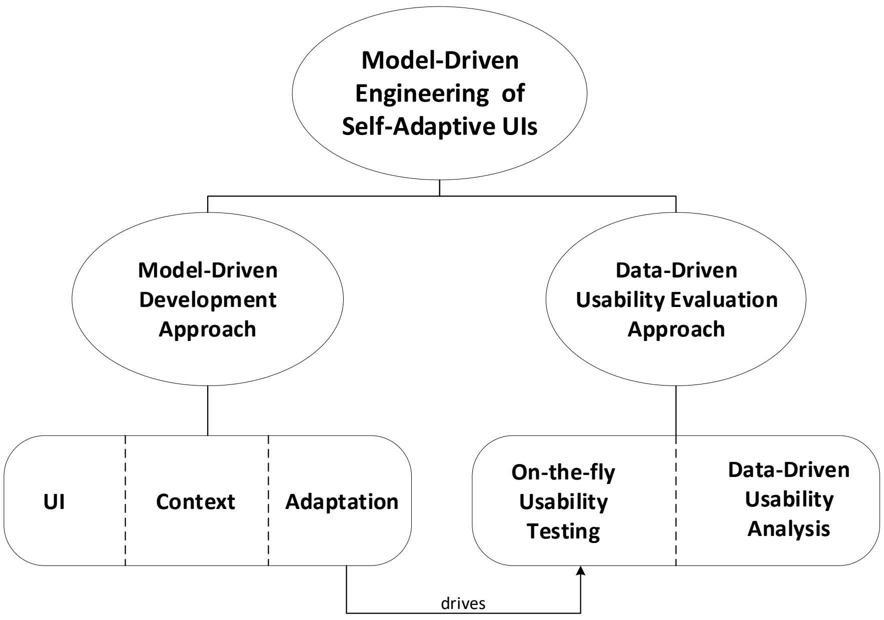

Model-Driven Engineering of Self-Adaptive UIs
Model-Driven Development Approach
Data-Driven Usability Evaluation Approach
UI
Context
Adaptation
On-the-fly Usability Testing
Data-Driven Usability Analysis
drives

Fig. 1.1 Abstract Solution Overview

Model-Driven Self-Adaptive UI Development Approach

Model-driven User Interface Development (MDUID) is a promising candidate and starting point for mastering the complex development task of self-adaptive UIs in a systematic, precise, and appropriately formal way. Our model-driven solution architecture for self-adaptive UIs is depicted in Figure 1.2 and consists of three development paths.

The first development path (left side of Figure 1.2) addresses the model-driven development of UIs. It makes use of an *Abstract UI Model* and a *Domain Model* which are then transformed by a code generator (*UI Generator*) into a *Final UI*. This development path has been subject of extensive research [PS12] and previous works present the realization and application of an MDUID approach for different target platforms ([YKUS16], [YS16b]) (including smartphone, desktop and self-service systems). The first development path supports efficient development of heterogeneous UIs for different target platforms. However, on its own, this development path is not enough to support context management and UI adaptation capabilities.

Therefore, in this thesis, we extended the classical MDUID approach with two additional parallel development paths which support model-driven context management and development of UI adaptations. In this way, the model-driven UI development path is complemented

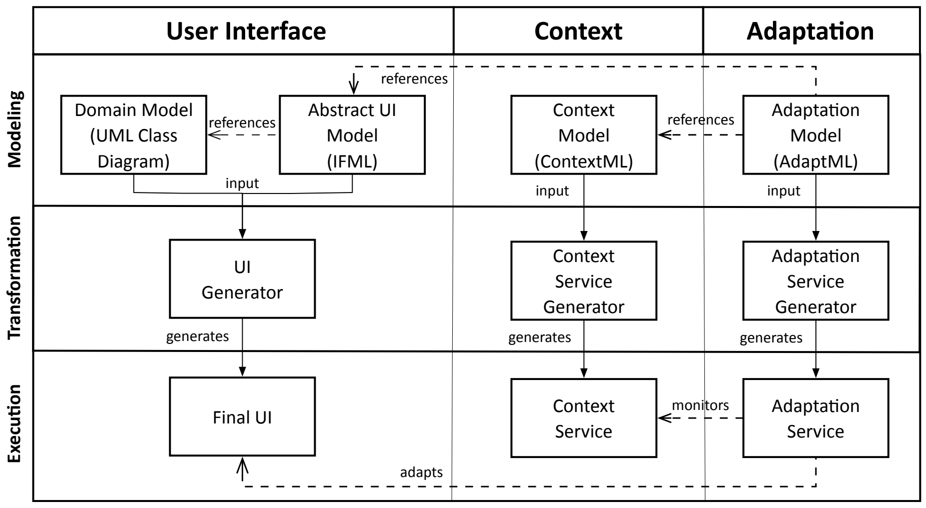


Fig. 1.2 Model-driven Architecture for Self-adaptive UIs

by analogous development paths responsible for context management and UI adaptation concerns. As these complementary paths are also based on the paradigm of model-driven development, the solution preserves various advantages of model-driven software development such as separation of concerns, extensibility or maintainability.

The second development path (in the middle of Figure 1.2) is responsible for characterizing the dynamically changing context-of-use parameters. A *Context Model* supports the abstract specification of heterogeneous context-of-use situations. Based on the *Context Model*, the *Context Service Generator* enables the generation of various *Context Services* which should monitor context information via hardware sensors.

The model-driven UI adaptation development path is depicted on the right side of Figure 1.2. In general, this development path supports the specification of an *Adaptation Model* in the means of abstract UI adaptation rules in alignment to an abstract UI modeling language. The UI adaptation rules, specified in the *Adaptation Model* reference the *Context Model* to define the context constraints for triggering adaptation rules. The UI adaptation rules of an *Adaptation Model* also have a reference to the *Abstract UI Model* to define which UI elements are scope of an UI adaptation change. The specified *Adaptation Model* serves then as an input for the *Adaptation Service Generator* which transforms it to an *Adaptation Service*. The *Adaptation Service* is responsible for monitoring the context information provided by the *Context Service* and adapting the generated *Final UI* at runtime.

For illustrating the interplay between the generated *Final UI*, the *Context Service*, and the *Adaptation Service* at runtime, as well as to present the effect of specified UI adaptation rules on the final user interface, we elaborate on the aspect of UI adaptation. Figure 1.3 shows a detailed overview of the runtime perspective for UI adaptation containing the main components for realizing self-adaptive UIs that are able to automatically react to changes in their context-of-use. Our solution architecture for realizing such self-adaptive UIs is based on IBM's MAPE-K [IBM05] architecture which is common in the field of self-adaptive software systems. In the following, the specific components of our solution architecture will be described.

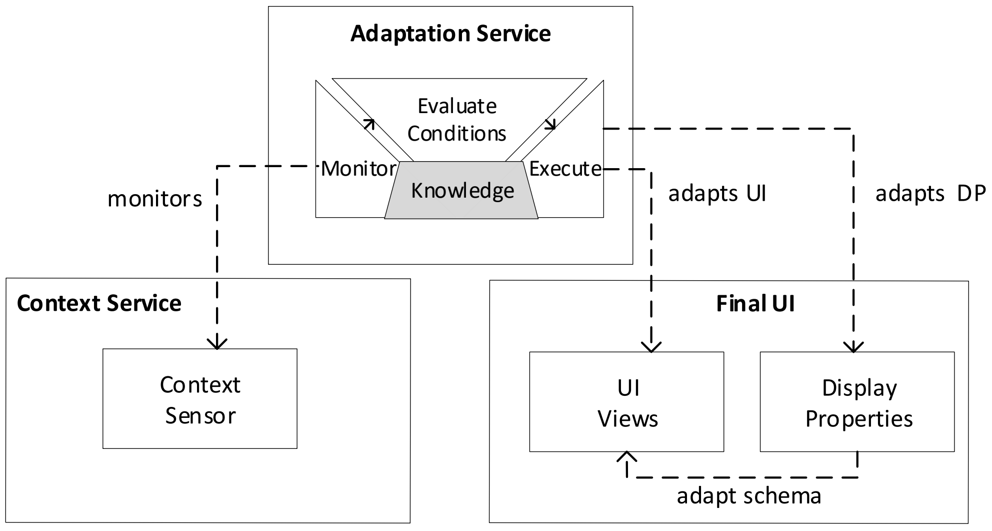


Fig. 1.3 Runtime perspective: Architectural Overview for Self-adaptive UIs

An essential component for UI adaptation at runtime is the *Context Service*. The *Context Service* provides context information to the *Adaptation Service* based on the *Context Sensors* which are specified in the *Context Model*. The provided context information is monitored by the *Adaptation Service*. Unlike the MAPE-K loop with its analyze and plan phases, the *Adaptation Service* relies on the application of predefined (by the abstract UI Adaptation Rules) conditions and associated actions. Hence, the two phases in the MAPE-K loop are replaced by the *Evaluate Conditions* component. The rules that satisfy the conditions are executed in the *Final UI*. The *Final UI* consists of two subcomponents: The *UI Views* which are responsible for representing the UI and *Display Properties* which are affected by the adaptation rules and contain the adaptable schema and type information of the UI. The executed adaptation operations can modify the UI directly or edit the *Display Properties*. In general, the UI is directly modified, if the change only affects the current view (adaptation of the current instance). If it is, for example, a property change that would affect several pages, it is set in the *Display Properties* (adaptation of schemas). An example for a property could be the layout of tables in the whole UI. The properties are referenced from within the views, and thereby can adapt the layout and design. The *Knowledge* component of the MAPE-K loop is responsible for storing different information during the monitoring and adaptation process. For example, monitored context information data, applied adaptation operations or user interaction data can be stored in the *Knowledge* component. In our approach, it is especially used for supporting the usability evaluation that is described in the following.

Data-Driven Usability Evaluation Approach

For addressing the challenges regarding usability evaluation of self-adaptive UIs, we developed a novel on-the-fly usability evaluation solution. The developed solution targets rule-based UI adaptation approaches and continuously monitors context information about context characteristics as well as collects instant user feedback about triggered UI adaptation features.

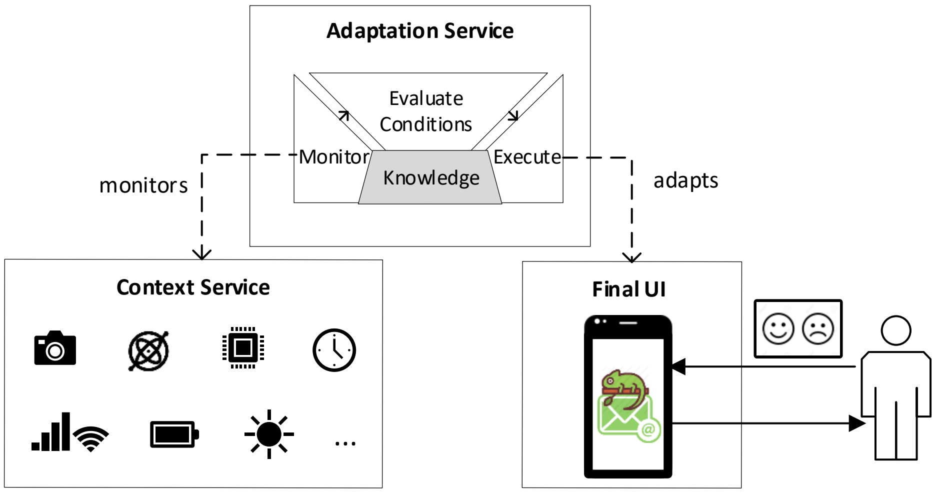


Fig. 1.4 Human-in-the-loop: On-the-fly usability evaluation of UI adaptation features

Figure 1.4 illustrates the main idea of our on-the-fly usability testing solution for UI adaptations. For supporting on-the-fly usability evaluation of UI adaptation features, we integrated an instant user feedback mechanism into the context monitoring and UI adaptation loop. The feedback mechanism allows users to explicitly rate the triggered UI adaptations. The users can give a positive or negative feedback or they can ignore that and focus on their main application task. As Figure 1.4 shows, there is a *Knowledge Base* which is responsible for storing all context information before and after a context change occurred (that lead to a UI adaptation triggering), all triggered adaptation rules, and the corresponding instant user feedback. Based on the stored information, it is possible to analyze the acceptance of UI adaptations based on the current context of the user and the user's feedback.

In summary, as depicted in Figure 1.5, this work provides the following contributions to support the different phases in model-driven engineering of self-adaptive UIs:

❶Modeling
Firstly, an integrated modeling approach is provided for *UI/Web Designers* to support the modeling of self-adaptive UIs. Two complementary modeling languages to the Interaction Flow Modeling Language (*IFML*) [Obj15] were developed to accompany the development process of self-adaptive UIs by explicitly covering the aspects of context management and UI adaptation. The first domain-specific language is called *ContextML* and supports the modeling of heterogeneous context-of-use parameters. The second developed domain-specific modeling language is called *AdaptML* and supports the specification of abstract UI adaptation rules that cover various UI adaptation techniques (e.g. layout, navigation, etc.).

❷Transformation
Beside the modeling phase, our model-driven engineering approach also supports the work of *Software Developers*. To reduce the amount of work, our engineering approach provides *Specific Code Generators* which enable the generation of the *Final UI* (FUI) code, as well as code for *Context Services* for monitoring context-of-use parameters and *Adaptation Services* to automatically adapt the UI at runtime. The *Specific Code Generators*, namely, *UI Generator*, *Context Service Generator*, and *Adaptation Service Generator* are basically realized by code generators that implement a model-to-text-transformation approach.

❸Execution
To enable the execution of generated *Context* and *Adaptation Services* in a flexible way, an execution environment for self-adaptive UIs is provided that makes use of a rule-based execution engine. The rule-based execution engine triggers UI adaptations based on identified context changes that might occur during the usage of the self-adaptive UI. The provided execution environment supports the work of *Software Developers* during the execution phase as the runtime behavior of the self-adaptive UI can be maintained in a flexible way by adding and removing *Context* and *Adaptation Services* as needed. Also, depending on the specific

usage context, the rule-based execution engine can be deployed and distributed over several client devices to enable a decentralized UI adaptation.

❹Evaluation and Application

Finally, our engineering approach incorporates a novel 'on-the-fly' usability evaluation solution for evaluating the acceptance of self-adaptive UIs. While the *End Users* are interacting with the interactive system during usage time, it allows to evaluate the *End User* satisfaction of self-adaptive UIs by combining context monitoring together with collection of instant user feedback. Furthermore, the benefit and applicability of our model-driven development approach is shown based on two application scenarios: self-adaptive UIs applied to a library and e-mail application.

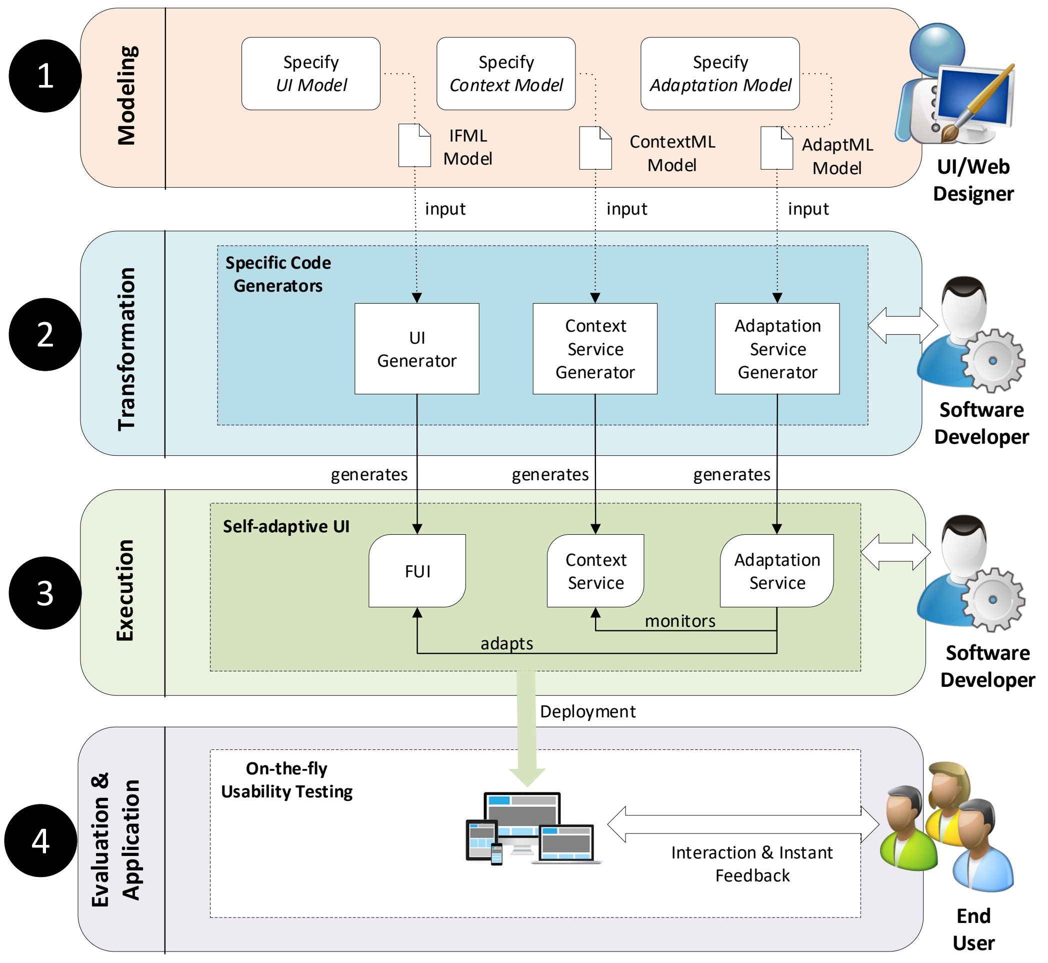


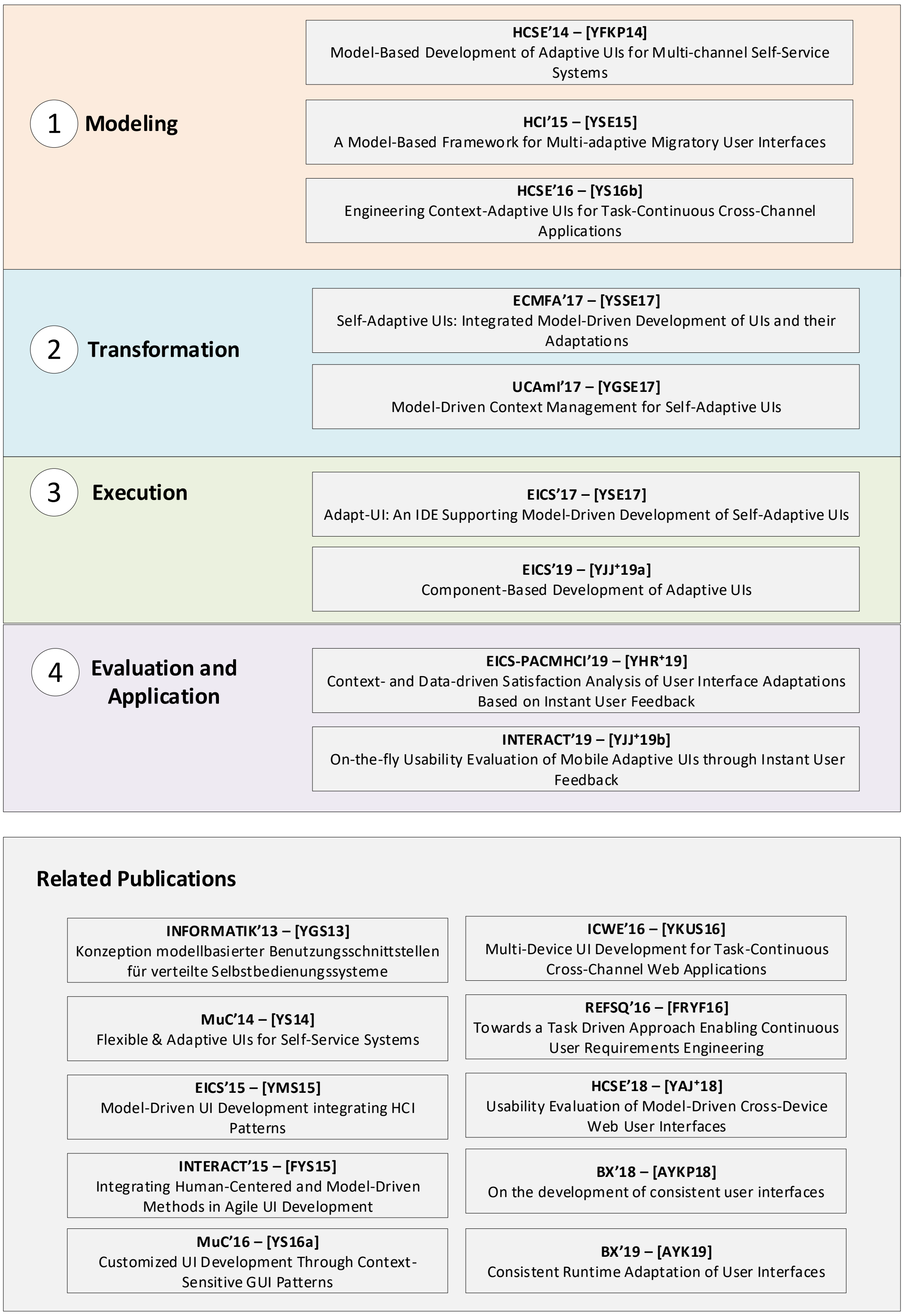
Fig. 1.5 Solution Overview

1.4 Publication Overview

Most of the presented contributions in this thesis have been reviewed and published in the proceedings of international conferences and workshops. An overview of these papers is given in Figure 1.6.

Our model-driven engineering approach for self-adaptive UIs emerged from an industrial research project in the context of model-driven UI development and Human-Computer Interaction (HCI). The research has been partly conducted in a joined project with Wincor Nixdorf International GmbH partially funded by the "‘it’s OWL"’ Leading-Edge Cluster of the German Federal Ministry of Education and Research (BMBF). The industrial project work has been performed within the s-lab - Software Quality Lab of the Paderborn University. In the following, the publications that were created in the course of this thesis and their influence on the solution will be briefly explained.

First of all, a number of publications is directly related to our solution. Initial ideas about a model-based UI development approach for distributed self-service systems in the context of the above mentioned joint industrial project with Wincor Nixdorf were presented in [YFKP14] (***HCSE’14 – [YFKP2014***]). In [YSE15] (***HCI’15 – [YSE2015***]), we presented a model-based reference framework for multi-adaptive migratory user interfaces. The presented framework supports the development of distributed UIs across a variety of devices and depicts a first runtime architecture for adaptive UIs. A refined solution and application of this framework is presented in our publication [YS16b] (***HCSE’16 – [YS2016a]***). In this work, we present a first state of our modeling languages and the model-driven engineering approach for adaptive UIs in the context of cross-channel applications. Beside these works on foundations and frameworks for adaptive UIs, we also focused on the specific requirements for supporting the different transformation steps in the solution to automatically derive code for self-adaptive UIs. The coupling of the MDUID process with a model-driven adaptation process was presented in [YSSE17] (***ECMFA’17 – [YSSE2017]***). In this work, we present our domain specific language for adaptation modeling *AdaptML* and how *Adaptation Services* can be generated based on the specified adaptation rules in order to automatically adapt the UI. The model-driven engineering approach for self-adaptive UIs was then completed by the complementary model-driven context management work [YGSE17] (***UCAmI’17 – [YGSE2017]***), where we introduced our context modeling language *ContextML* and showed analogously the generation of *Context Servi*ces for monitoring the context-of-use parameter which trigger the UI adaptations at runtime. Furthermore, in our work [YSE17] (***EICS’17 – [YSE2017]***) we presented our tool-support for model-driven engineering of self-adaptive UIs. The presented tool-chain supports developers in the different phases of the engineering process from modeling, transformation to execution of self-adaptive UIs. Based on our



1 Modeling

HCSE'14 – [YFKP14]
Model-Based Development of Adaptive UIs for Multi-channel Self-Service Systems

HCI'15 – [YSE15]
A Model-Based Framework for Multi-adaptive Migratory User Interfaces

HCSE'16 – [YS16b]
Engineering Context-Adaptive UIs for Task-Continuous Cross-Channel Applications

2 Transformation

ECMFA'17 – [YSSE17]
Self-Adaptive UIs: Integrated Model-Driven Development of UIs and their Adaptations

UCAmI'17 – [YGSE17]
Model-Driven Context Management for Self-Adaptive UIs

3 Execution

EICS'17 – [YSE17]
Adapt-UI: An IDE Supporting Model-Driven Development of Self-Adaptive UIs

EICS'19 – [YJJ+19a]
Component-Based Development of Adaptive UIs

4 Evaluation and Application

EICS-PACMHCI'19 – [YHR+19]
Context- and Data-driven Satisfaction Analysis of User Interface Adaptations Based on Instant User Feedback

INTERACT'19 – [YJJ+19b]
On-the-fly Usability Evaluation of Mobile Adaptive UIs through Instant User Feedback

Related Publications

INFORMATIK'13 – [YGS13]
Konzeption modellbasierter Benutzungsschnittstellen für verteilte Selbstbedienungssysteme

ICWE'16 – [YKUS16]
Multi-Device UI Development for Task-Continuous Cross-Channel Web Applications

MuC'14 – [YS14]
Flexible & Adaptive UIs for Self-Service Systems

REFSQ'16 – [FRYF16]
Towards a Task Driven Approach Enabling Continuous User Requirements Engineering

EICS'15 – [YMS15]
Model-Driven UI Development integrating HCI Patterns

HCSE'18 – [YAJ+18]
Usability Evaluation of Model-Driven Cross-Device Web User Interfaces

INTERACT'15 – [FYS15]
Integrating Human-Centered and Model-Driven Methods in Agile UI Development

BX'18 – [AYKP18]
On the development of consistent user interfaces

MuC'16 – [YS16a]
Customized UI Development Through Context-Sensitive GUI Patterns

BX'19 – [AYK19]
Consistent Runtime Adaptation of User Interfaces

Fig. 1.6 Publication Overview

tool-chain, in [YJJ+19a] (***EICS'19 – [YJJKAE2017]***) we presented an additional solution for development of adaptive UIs which complements the model-driven engineering approach through a component-based development approach. In this work, the generated *Context* and *Adaptation Services* were encapsulated as independent software components to improve reusability. For the evaluation of self-adaptive UIs, we have developed a novel on-the-fly usability evaluation method which was presented in [YHR+19] (***EICS-PACMHCI'19 – [YHMASE2019]***). A demo of this evaluation method was presented in [YJJ+19b] (***INTER-ACT'19 – [YJJSE2019]***).

Beside peer reviewed international conference papers directly related to the core solution of this thesis, several other papers were published that are related to the fields of MDUID, usability engineering, and consistent UI adaptation. An overview of these related publications is shown at the bottom of Figure 1.6.

1.5 Thesis Structure

An overview of the structure of this thesis is shown in Figure 1.7.

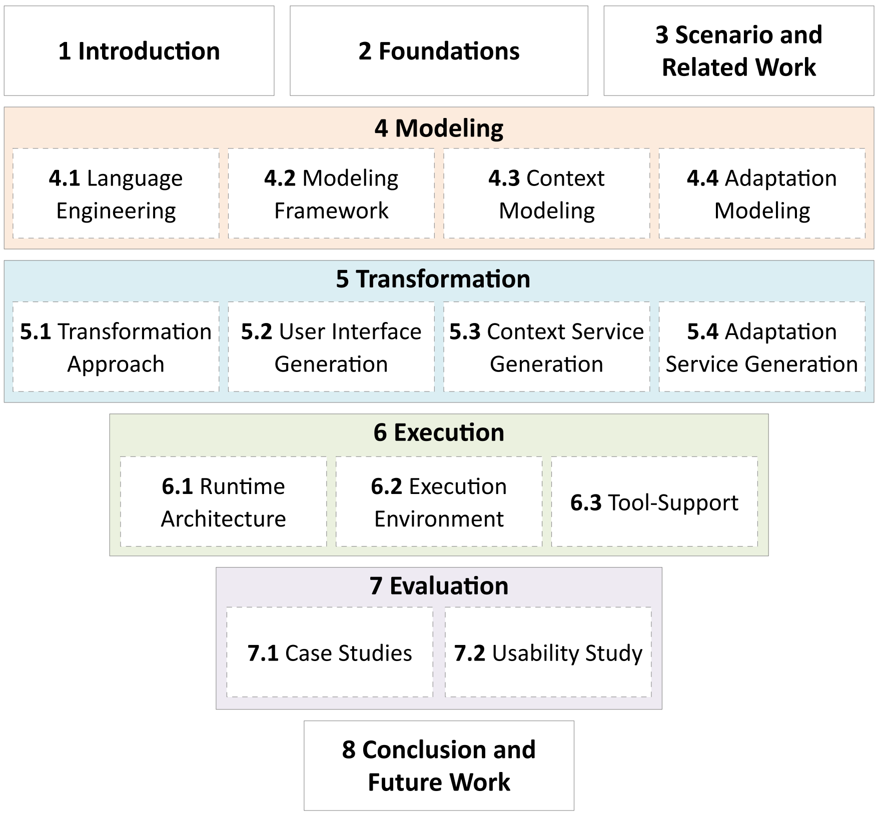


Fig. 1.7 An Overview of the Thesis Structure

As can be seen, the remainder of this thesis is structured as follows:

- Chapter 2 lays the foundations for the presentation of our approach. We address the general concepts of model-driven software development while especially focusing on model-driven user interface development and UI modeling languages. Further, we present important foundations on context-aware and self-adaptive software systems and introduce relevant aspects of adaptive user interfaces.

- In Chapter 3, we present a detailed problem analysis by comparing and discussing existing state-of-the-art approaches. Based on this discussion, we depict open issues in this research area and derive concrete requirements for solving the issues.

- Our modeling approach for engineering self-adaptive UIs is described in Chapter 4. This chapter introduces first the integrated modeling framework for self-adaptive UIs. After that, we present our modeling language *ContextML* for describing contextual parameters representing the different contexts-of-use. Furthermore, our adaptation modeling language *AdaptML* for specifying UI adaptation rules is described.

- In Chapter 5, we present our transformation approach. In this context, we first give an overview on the overall transformation approach. After this, we describe the needed transformation steps for generating self-adaptive UIs. For this purpose, we describe in detail how code for the final UI as well as *Context* and *Adaptation Services* are generated based on the corresponding models.

- A description of our rule-based execution environment is presented in Chapter 6. In this chapter, we first give an overview on the runtime architecture for self-adaptive UIs. After that, we describe our implemented UI execution environment that is responsible for runtime adaptation of the generated self-adaptive final UIs. Furthermore, we present the developed tool-chain for supporting model-driven development of self-adaptive UIs.

- In Chapter 7, the evaluation of our approach is described. For this purpose, we first describe two case-studies which show the application of our engineering approach to devise self-adaptive UIs. Then, we present our on-the-fly usability evaluation solution which is used to assess the end-user satisfaction of self-adaptive UIs. Finally, we describe and discuss the gained results of our usability experiment based on a data-driven usability evaluation.

- We conclude this thesis in Chapter 8. We summarize the contributions of our approach and give an outlook of future work and research challenges.

Chapter 2

Foundations

In this chapter, we give an overview of foundations that are relevant for this thesis. First, we introduce in Section 2.1 the general concepts of model-driven user interface development. In Section 2.2, we present the main idea of user interface description languages in the context of model-driven UI development and give an overview of the Interaction Flow Modeling Language (*IFML*) which is an OMG standardized UI modeling language. Further, we introduce general concepts of context-aware computing and context modeling approaches in Section 2.3. In Section 2.4, we present relevant concepts of self-adaptive software systems and introduce self-adaptive user interfaces. In Section 2.5, we describe relevant concepts and methods related to usability engineering. Finally, Section 2.6 gives an overview of used technologies.

2.1 Model-Driven User Interface Development

This section provides basic information about model-driven user interface development (MDUID) and its background. The main goals and concepts of MDUID are explained. In addition, two related conceptual approaches, the Model Driven Architecture (MDA) and the CAMELEON Reference Framework (CRF), are introduced.

2.1.1 Background

For describing the underlying concepts of model-driven UI development, we start with a historical background by differentiating between model-based and model-driven software development.

Model-based development (MBD) assumes the use of models during development. In this context, models are primarily used to facilitate the communication between experts from different domains. However, the models are not directly involved in the development process and are thus an operational add-on [BCW12]. Model-based development has been used for UI development since the 1980s ([Sze96], [MPV11]), aiming for high-quality UIs with reduced development effort. In the 1990s, model-based UI development approaches already specified architectures, development methods based on models, and how these models have to be refined to achieve running applications, in addition to exploiting their communicative value (e.g., Mobi-D [Pue97] or Trident [BHV^{+}94]).

Model-driven development (MDD) is the general approach of making models the primary artifact in the development process rather than application code, along with automated model transformations ([Sch06], [BCW12]). Models are described by a modeling language, and a modeling language in turn is defined by a so-called meta-model. An MDD process usually involves multiple models on different levels of abstraction. Through model transformations, defined by transformation rules, high-level models can be stepwise transformed into more concrete models and finally into implementation code. Model transformation rules are defined on the meta-model level hence they can be applied for any model that is an instance of the meta-model. Since MDD has been established in software engineering, several powerful tools and concepts which support the work with models, meta-models and model transformations are available. Some of them are discussed in the later sections of this chapter. The core idea of MDUID is to apply the principles mentioned above on user interface development. Based on models and model transformation, MDUID has the potential to improve user interface development by the following four characteristics (taken from [MPV11]):

- *Multiple levels of abstraction*: The user interface is described on multiple levels of abstraction by models. Usually a certain abstraction level abstracts from a concrete aspect, e.g., a specific technology or the final user interface design.
- *Reusability*: As high-level models and meta-models are independent from certain aspects such as platform, technology and/or user interface design, they naturally have a reusability value.
- *Machine readability*: Model instances are limited by the formal definition of their related modeling language, thus they have a specific format. Formatted data can be processed by a computer system. This is a basic precondition for automated transformations of model instances.

- *Automated transformation*: Model transformations describe mappings between a source model and a target model or executable/interpretable code. Once such a mapping is implemented, it can be automatically executed for any instance of the source model.

Model-driven development supports the separation of software into different abstraction levels, each with its own purpose. This enables the usage and creation of models describing the user interface at different phases of the development process. Even non-technical stakeholders can participate in the early stage of interaction and interface design. This in turn results in a development process where requirements defined by stakeholders and domain experts can be validated early on. Another well-known use case is the separation of content of user interfaces from the design of user interfaces. In such a way, content can be reused in combination with different designs. Beside the separation of software into different abstraction levels, model driven development also enables the separated development of a software system according different development aspects such as UI, core application or data management. The applicability of such a separated model-based/-driven development of UIs complementary to the core application and data management is shown in [Bot11]. However, the implementation of an MDUID process, in general, is a time-consuming and expensive task, including the definition of multiple modeling languages, model transformations, and appropriate tooling if needed [BCW12].

2.1.2 Model Driven Architecture

A specific type of MDD and an approved standard in the field of software engineering is the Model Driven Architecture (MDA), which was proposed by the OMG in 2003 (see [MM03]). The core idea of MDA is similar to MDD, namely to abstract from the plain application code to a more general and less complex representation, i.e., models. Beside documentation and specification purpose, MDA further suggests models for the definition of the architecture, design, and implementation of software. Therefore, in MDA, three model levels and transformations between them are specified. As depicted in Figure 2.1, MDA differentiates between the *Computation-Independent Model (CIM)*, *Platform-Independent Model (PIM)*, and *Platform-Specific Model (PSM)* level. The objective of these model levels is the so-called separation of concern. PIM and PSM are the core models. On the CIM level, the software is described by its general requirements. On the PIM level, the software is described in terms of a software specification (users point of view) while on the PSM level the software is described in terms of the technical realization (developers point of view) including aspects of a concrete platform. This clear separation of platform-independent and platform-dependent information facilitates reusability and portability of PIM instances [Pet07], since they can be maintained for other platforms, PSMs.

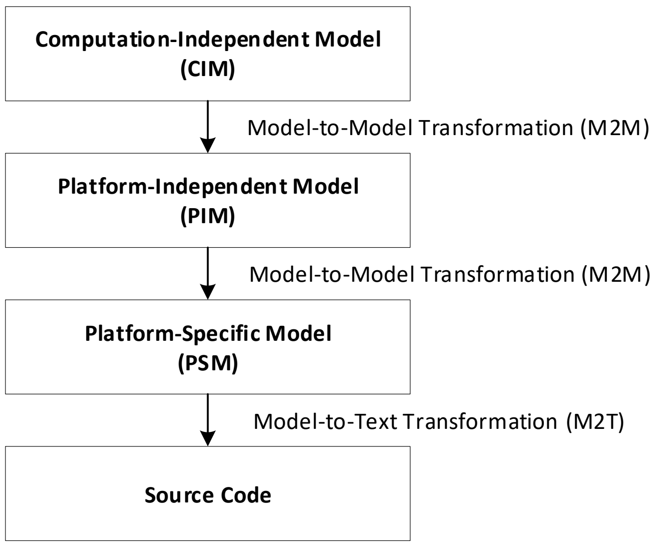

Computation-Independent Model (CIM)
Model-to-Model Transformation (M2M)
Platform-Independent Model (PIM)
Model-to-Model Transformation (M2M)
Platform-Specific Model (PSM)
Model-to-Text Transformation (M2T)
Source Code

Fig. 2.1 Model Driven Architecture (MDA)

Furthermore, MDA suggests the transformations of model instances between model levels and the generation of application code from PSM instances. These *Model-to-Model (M2M)* and *Model-to-Text (M2T)* transformations are represented by arrows in Figure 2.1. The last transformation from PSM to implementation is usually realized by an automated M2T transformation while inter-model transformations from PIM to PSM are realized by M2M transformations. M2T transformations define, which code snippets are generated for the different PSM elements. Since the transformations are defined on the model level, they can be reused for any instance of the PSM and the PIM. Another principle of MDA is the concept of meta-models. Meta-models are models of modeling languages, i.e., they facilitate the definition of a modeling language itself. MDA provides the general concept for separating specification and technical realization of software, *Model-to-Model* and *Model-to-Text* transformations, and metamodeling. Particularly, interactive applications with complex user interfaces have more specific requirements than software in general. Therefore, a framework is introduced in the following chapter that applies the principles of MDA to model-driven development of interactive applications.

2.1.3 CAMELEON Reference Framework

While MDA has been designed as a general-purpose architecture for model-driven software development, the CAMELEON Reference Framework [CCT+03], translates the conceptual approach of MDA to the model-based/-driven UI development domain.

The CAMELEON Reference Framework (CRF) is a result of the FP5 CAMELEON (Context Aware Modeling for Enabling and Leveraging Effective interactiON) project carried out by a consortium composed of various universities and research groups. It was conceived as a conceptual framework which supports the model-based development process of multi-target user interfaces. However, the CRF is not a prescription of methods and procedures which describe how the different steps of development can be realized. The framework rather provides a unified understanding and common representation of user interface development and its related models, methods and processes. In order to do so, the CRF defines a conceptual reference framework composed of different abstraction layers, which are important for model-based/-driven UI development, and the relationships between them.

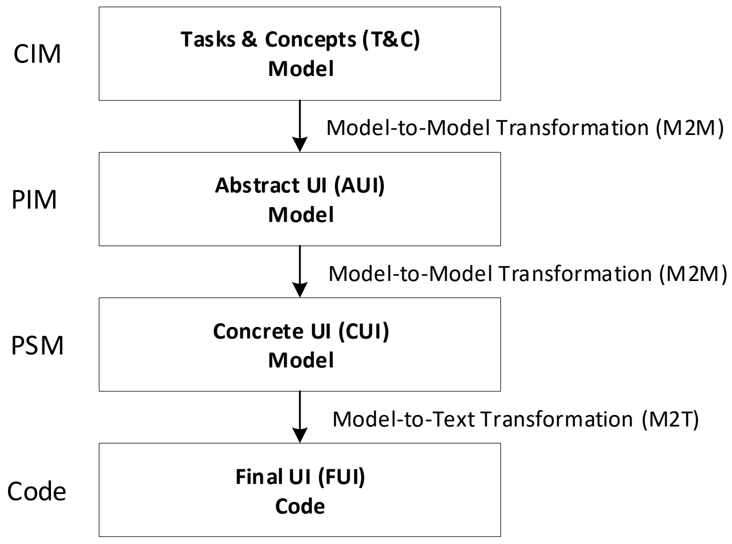

CIM
Tasks & Concepts (T&C) Model
Model-to-Model Transformation (M2M)
PIM
Abstract UI (AUI) Model
Model-to-Model Transformation (M2M)
PSM
Concrete UI (CUI) Model
Model-to-Text Transformation (M2T)
Code
Final UI (FUI) Code

Fig. 2.2 CAMELEON Reference Framework (simplified)

As illustrated in Figure 2.2, the CRF consists of four basic layers. The first abstraction layer is the *Tasks and Concepts Model*. It defines the tasks that can be performed by the user during interaction with the user interface. It also considers the hierarchy of these tasks and their temporal order. In addition to such a *Task* model, the *Tasks and Concepts Model* layer can contain a *Domain* model. The Domain model describes the information handled by the application, e.g., the data types of properties or methods accessed by the user interface.

The *Abstract User Interface (AUI) Model* expresses the user interface by means of abstract containers and individual components. While containers describe logical groupings, individual components are mainly input/output definitions or actions performed on these inputs/outputs. Both containers and components are Abstract Interaction Objects (AIOs), i.e., they are independent of any platform or interaction type available on the target (e.g., graphical, vocal, video-based, virtual [Van08] p.4). In other words, the AUI model only contains information about what is presented and not how it is presented.

The next layer, *Concrete User Interface (CUI)*, is derived by transforming the AUI model into an interactor-dependent representation, i.e., AIOs are reified into Concrete Interaction Objects (CIOs). CIOs depend on a specific interactor type, but are still implementation-language independent. A specific interactor type is for example a Button or a Voice Command. CIOs clarify how the interface is perceived by the users. This includes layouting, coloring and the actual presentation of the user interface elements. Typical examples of CIOs are frames, buttons or voice entries.

The *Final User Interface (FUI)* layer represents the UI by source code depending on one or more specific implementation technologies. It can be expressed in any implementation language, e.g., Java or a mark-up language such as XML. Usually, the FUI is generated from the CUI model and can be interpreted or compiled, depending on the applied implementation technology.

As shown in Figure 2.2, the CRF suggests a four-step transformation process from high-level abstract descriptions to implementation code. Starting with the *Tasks and Concepts* model, an AUI model is derived and in turn transformed to a CUI model. While these transformation steps are usually supported by *Model-to-Model* transformations, the last step to the FUI is realized by a *Model-to-Text* transformation.

Model-based and model-driven UI development approaches require UI models for specifying the UI on different abstraction levels of the CAMELEON Reference Framework (CRF). Therefore, the following subsection deals with the topic of *User Interface Description Languages (UIDLs)* to describe the means for characterizing UI aspects based on models.

2.2 User Interface Description Languages (UIDLs)

This subsection provides an overview of common user interface description languages (UIDLs) in the context of MDUID. Moreover, the *Interaction Flow Modeling Language (IFML)* is presented which plays an important role in this thesis.

2.2.1 Overview

Model-based and model-driven UI development approaches require UI models for specifying the UI on different abstraction levels of the CAMELEON Reference Framework (CRF). For this purpose, different User Interface Description Languages (UIDLs) exist. Regarding the *Tasks and Concepts* layer, different types of Task models and Domain models are used. For describing the tasks of an interactive system, task-tree based modeling approaches such as ConcurTaskTrees (CTT) [PMM97] or HAMSTERS [PM15] are widely spread. On the same abstraction level, Domain models like UML class diagrams are often used to characterize the relevant data entities for the interactive application. For describing the AUI layer, abstract UI modeling languages such as MARIA XML [PSS09], DISL [SBM06] or the OMG standard *IFML* [BF14] were introduced. Furthermore, there are also UI modeling languages for the CUI layer such as UIML [APB+99], MARIA XML [PSS09] or UsiXML [LVM+04]. Through its mobile and web extension mechanisms (details can be found in the next subsection) *IFML* is also supporting modeling of CUI aspects. Please note that UsiXML and MARIA XML are UI modeling languages that build up on CTT and cover all abstraction layers including AUI and CUI layers to finally derive code for the final UI. Although both of the UI modeling languages UsiXML and MARIA XML are widely studied and applied in different application areas, in this thesis we focus and build up on the *IFML* as it is an OMG standardized language for UI modeling. In the following, *IFML* is introduced and presented in more detail.

2.2.2 IFML

The Interaction Flow Modeling Language (*IFML*) is an abstract user interface modeling language and was adopted as a standard by the OMG in March 2013 [BF14]. *IFML* is a UML profile and originates from the Web Modeling Language (WebML) [CFB00], which was defined in 2000. While WebML focuses on the conceptual definition of data-intensive Web applications, *IFML* is designed for a more general usage. As its name suggests, *IFML* works with interaction flows and interaction flow elements, which describe the content, the user interaction, and the control behavior of any application rather than only web applications. In other words, *IFML* supports the specification of a user interface's general structure, its content, events, event handlers, and input/output parameters which are bound to specific view components ([BF14] pp.5-6).

Currently, *IFML* is supported by two editor tools. The first is the WebRatio[1] development platform, a proprietary tool supporting MDD as well as the integration of external systems and services in combination with *IFML*. The second tool is an open-source editor[2] based on the Sirius framework and is currently in development state. The Sirius framework is a framework with which workbenches providing editors including diagrams, tables or trees can be defined and deployed into Eclipse IDEs. Such workbenches provide rich and specialized modeling editors that support the creation, deletion, modification of according models.

An increasing number of business-to-consumer (B2C), business-to-business (B2B), and business-to-employee (B2E) applications rely on browser-based GUIs with capabilities such as form-based interaction, information browsing and link navigation. Usually, these GUIs are built on top of a variety of technologies and platforms. As described in [BF14], *IFML* is motivated by such applications. It supports the platform-independent description of an application's front-end as it is perceived by the end user. As a PIM language, *IFML* follows the separation of concern paradigm and perfectly fits into the AUI level of the CRF [RMB13]. Furthermore, *IFML* includes a well-defined specification as well as a standardization document providing guidelines for the mapping of *IFML* models to PSMs.

IFML metamodel and basic modeling concepts

The Interaction Flow Modeling Language (IFML) is designed for expressing the content, user interaction and control behavior of the front-end of software applications. Its metamodel uses the basic data types from the UML metamodel, specializes a number of UML metaclasses as the basis for IFML metaclasses, and presumes that the Domain model is represented in UML. A simplified excerpt of the IFML metamodel, based on [BUA17], is depicted in Figure 2.3 which illustrates the main *InteractionFlowElements* and *InteractionFlows*.

In the following, the relevant *IFML* concepts used in this thesis and their corresponding notation in the *IFML* visual syntax are presented. This description is based on the modeling notation description in the *IFML* specification document.

ViewContainer

An important *InteractionFlowElement* in *IFML* to support view structure specification is a *ViewContainer* (specialization of *ViewElement*, see Figure 2.3) which consists of one or more possibly nested *ViewContainers*. For example, windows in traditional desktop applications or page templates in web applications. A *ViewContainer* is a user interface

[1]https://www.webratio.com/site/content/en/home

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

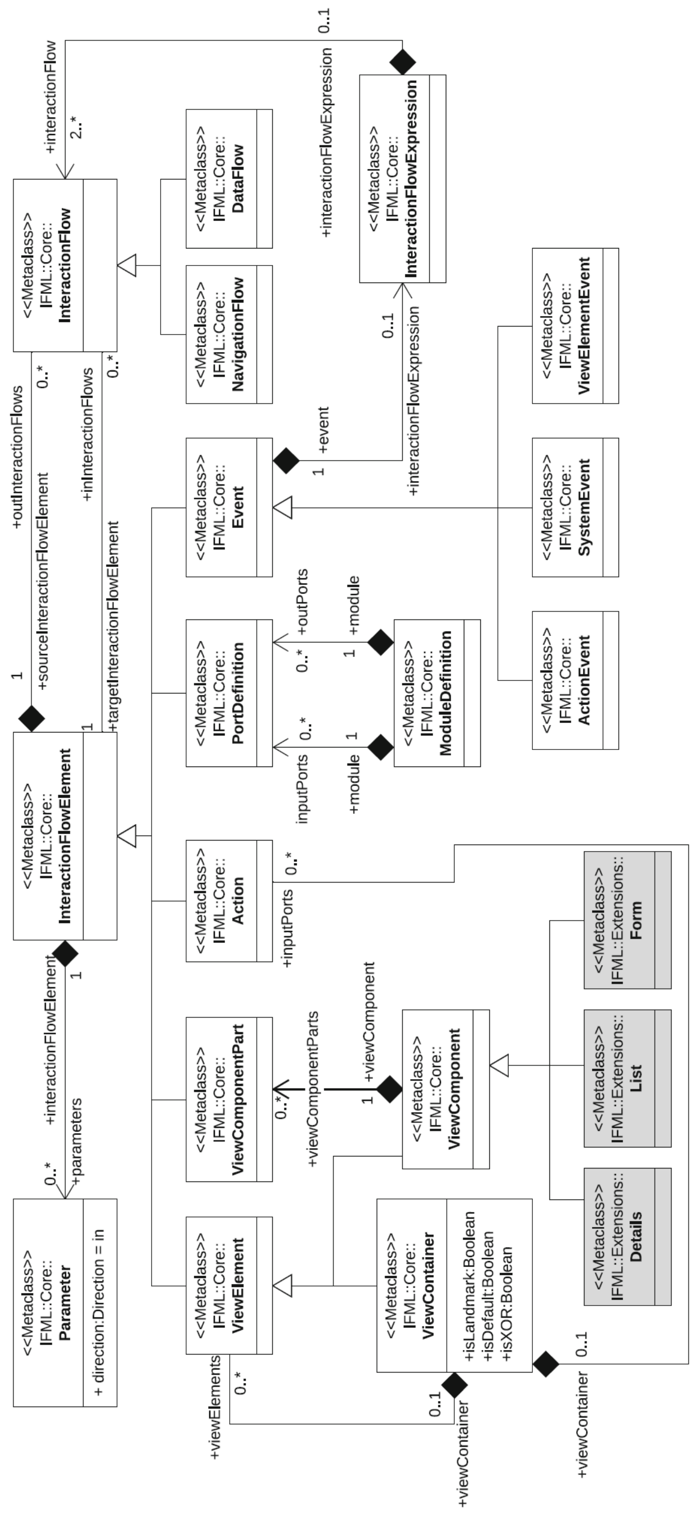

<<Metaclass>> IFML::Core:: InteractionFlow
<<Metaclass>> IFML::Core:: NavigationFlow
<<Metaclass>> IFML::Core:: DataFlow
<<Metaclass>> IFML::Core:: InteractionFlowExpression
<<Metaclass>> IFML::Core:: InteractionFlowElement
<<Metaclass>> IFML::Core:: Event
<<Metaclass>> IFML::Core:: PortDefinition
<<Metaclass>> IFML::Core:: ModuleDefinition
<<Metaclass>> IFML::Core:: SystemEvent
<<Metaclass>> IFML::Core:: ViewElementEvent
<<Metaclass>> IFML::Core:: ActionEvent
<<Metaclass>> IFML::Core:: Action
<<Metaclass>> IFML::Core:: ViewComponentPart
<<Metaclass>> IFML::Core:: ViewComponent
<<Metaclass>> IFML::Core:: ViewElement
<<Metaclass>> IFML::Core:: ViewContainer
+isLandmark:Boolean
+isDefault:Boolean
+isXOR:Boolean
<<Metaclass>> IFML::Core:: Parameter
+ direction:Direction = in
<<Metaclass>> IFML::Extensions:: Form
<<Metaclass>> IFML::Extensions:: List
<<Metaclass>> IFML::Extensions:: Details
+interactionFlow
2..*
0..1
+interactionFlowExpression
+outInteractionFlows
0..*
+sourceInteractionFlowElement
1
+inInteractionFlows
0..*
+targetInteractionFlowElement
1
+event
1
+interactionFlowExpression
0..1
+outPorts
0..*
+module
1
inputPorts
0..*
+module
1
+inputPorts
0..*
+interactionFlowElement
1
+parameters
0..*
+viewComponentParts
0..*
+viewComponent
1
+viewElements
0..*
+viewContainer
0..1
+viewContainer
0..1

Fig. 2.3 An excerpt of the *IFML* metamodel based on [BUA17]

element which contains other interface elements (other *ViewContainers* or *ViewComponents*). It can group elements that can be accessed by the user at the same time, and it can also do the opposite: grouping elements which the user can only see alternatively. Figure 2.4 depicts the four different types of *ViewContainers*. Figure 2.4(a) shows a basic *ViewContainer* that can group other user interface elements. Figure 2.4(b) shows an XOR *ViewContainer*, denoted by '[XOR]'. XOR *ViewContainers* group other *ViewContainers* and constrain them to be displayed alternatively. Only one *ViewContainer* inside an XOR *ViewContainer* can be shown at a time. When an XOR *ViewContainer* is displayed, the container's default contained *ViewContainer* will be displayed. Such a default *ViewContainer* is depicted in Figure2.4(d) and it is denoted by '[D]'. Another type of *ViewContainer* is landmark, depicted in Figure 2.4(c). Landmarks, by definition, can be reached from any other interface element. Landmarks are denoted by '[L]'.

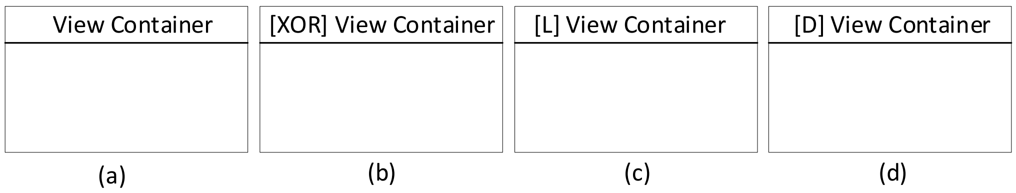


Fig. 2.4 Different types of *View Containers* in *IFML*

ViewComponent

A *ViewComponent*, as depicted in Figure 2.5(a), is a user interface element that is contained in a *ViewContainer*. It can present content to the user and/or allows for interaction. *ViewComponents* denote the publication of static or dynamic content, or interface elements for data entry (such as input forms). A *ViewComponent* can have input and output parameters. An example for a *ViewComponent* is a list of search result items or a contact form.

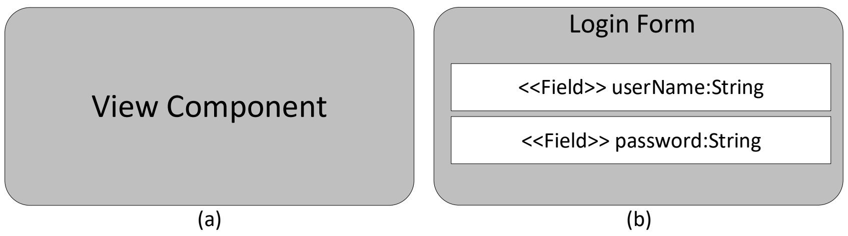


Fig. 2.5 *ViewComponents* and *ViewComponentParts* in IFML

A *ViewComponentPart* is an element that can only reside in a *ViewComponent* (see Figure 2.5(b)). It can provide more in-depth interaction details of a *ViewComponent*. For example, a login form that contains a password input field, where the form itself is the *ViewComponent* and the password field the *ViewComponentPart*.

Event

A *ViewContainer* and a *ViewComponent* can be associated with an *Event* (see Figure2.6), that can represent users' interactions or system-generated occurrences. For example, an *Event* for selecting one or more items from a list can be characterized through a *Select Event* or for submitting inputs from a form through a *Submit Event*. A *Throwing Event* describes a system generated event that can be used to trigger an *Action*.

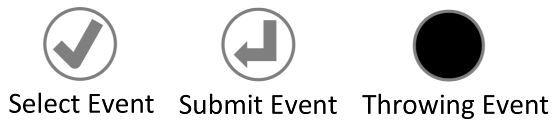


Fig. 2.6 *Events* in IFML

Action

An *Action* in *IFML* as shown in Figure 2.7 is an operation performed by the application behind the scenes. *Actions* are triggered by *Events*. For example, delete an item from a user's profile on request of the user (the user request is the triggering event).

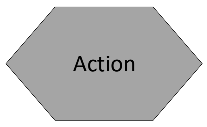


Fig. 2.7 *Action* element in IFML

InteractionFlow

The effect of an *Event* is represented by an *InteractionFlow* connection. The *InteractionFlow* expresses a change of state of the user interface. An *InteractionFlow* carries parameters between elements in the *IFML* model, upon the occurrence of an *Event*. The parameters sent in the outgoing side of the flow are used as input parameters for the element at the incoming side of the flow. As depicted in Figure 2.8, there are two kinds of *Interaction Flow*: *NavigationFlow* and *DataFlow*. *NavigationFlows* (denoted by a full arrow line) do not only carry the parameters, but also navigate the user to the element at the arriving side of the flow. *DataFlows* (denoted by a dashed arrow line), on the other hand, are only used to carry parameters around. They are often used as auxiliary flows for *NavigationFlows* when not all the necessary parameters can be found in the *IFML* element at the outgoing side of the *NavigationFlow*. Also worth mentioning is that *NavigationFlows* do not necessarily need to carry parameters (e.g., when the user browses to another page in a website, merely following a hyperlink). An example of a *NavigationFlow* which does carry parameters can be the following: a user submits an item using a form and arrives at a details page of the item he just inserted (carried parameters: the details of the item).

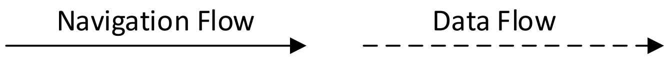


Fig. 2.8 *InteractionFlows* in IFML

Parameter

A *Parameter* as depicted in Figure 2.9(a), is a variable with a typed and named value that can be passed around by flows. It can be held by any *IFML* element which can have incoming or outgoing flows. For example, a *View Component* that contains a list of songs can contain the *Parameter* SelectedSong. When the user selects a song from the list, the parameter is set and can be passed onto a flow which leads to another interface element playing the selected song.

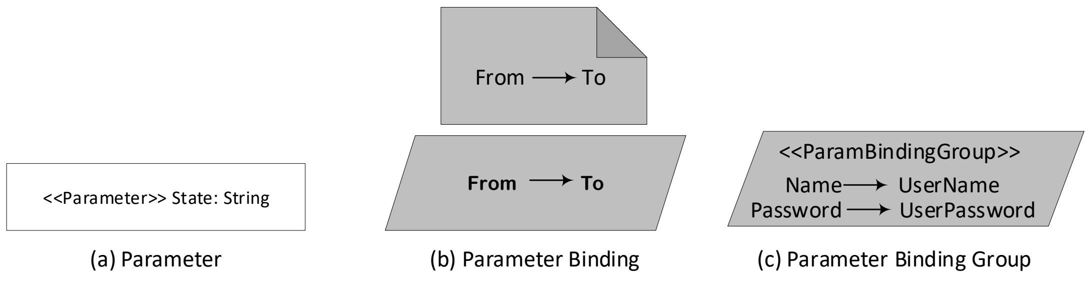


Fig. 2.9 *Parameters* in IFML

A *ParameterBinding* as shown in Figure 2.9(b), is the association of input and output parameters of a flow. They specify how parameters are transferred. For example, the *Parameter* SelectedSong of a list with songs is bound to the Song parameter of a details view about the song, upon transferring it between both views by means of a navigation flow. *ParameterBindings* have two syntactical notations with equal semantics, and are connected to the associated flow by means of a dashed line.

Figure 2.9(c) depicts a *ParameterBindingGroup* which groups several *ParameterBindings* which are associated to the same flow. For example, to login to a website both parameters username and password are sent to an action (at the server-side) which checks whether the username-password combination is valid. In this example, both username and password are bound in the *ParameterBindingGroup* of the flow between the login form and the server action.

IFML Example

Beside the above described basic *IFML* modeling concepts there are further concepts such as *Modules* for compact representation of complex *IFML* models, *ActivationExpressions* for defining interaction flow constraints, and also many other extension mechanisms for modeling specific front-end aspects of mobile and web applications. The interested reader may refer to the *IFML* book published by Brambilla et al. [BF14]. Instead of further elaborating on the modeling concepts of *IFML*, we present, in the following, a small *IFML* modeling example to provide a better intuition for the usage of *IFML*. Figure 2.10 shows an initial example *IFML* model of a simple user interface. The view structure consists of three *ViewContainers*

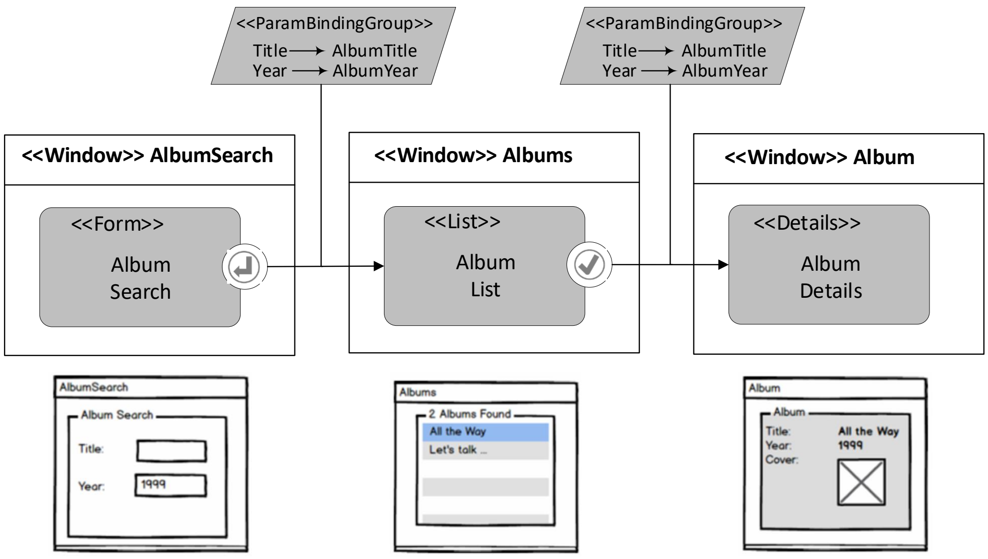
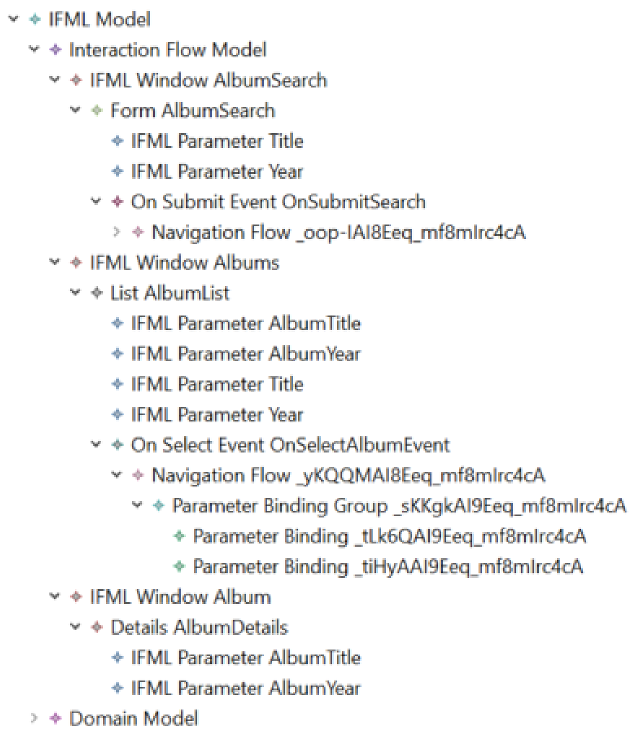


Fig. 2.10 Example of a simple UI and its *IFML* specification.

(`AlbumSearch`, `Albums`, and `Album`), which reflect the top-level organization of the graphical UI in three distinct pages. The model shows the content of each *ViewContainer*. For example, the `AlbumSearch` *ViewContainer* comprises one *ViewComponent* called `AlbumSearch`. This notation represents the content of the respective page in the GUI (i.e., a form to search for a specific book based on the title and publication year). *Events* are represented in *IFML* as circles. The `SubmitEvent` *Event* specifies that the `AlbumSearch` component is interactive and it triggers the *Parameter* passing from the *ViewComponent* owning the *Event* to the *ViewComponent* target of the *NavigationFlow* outgoing from the *Event*. In the graphical UI, it means that the user can enter values for the search regarding title and year. The effect of the `SubmitEvent` *Event* is represented by the outgoing arrow (called *InteractionFlow* in *IFML*), which specifies that the triggering of the *Event* causes the display of the `Albums` *ViewContainer* and the display of its `AlbumList` *ViewComponent* (i.e., the list of albums based on the search results). The input–output dependency between the `AlbumSearch` and the `AlbumList` *ViewComponents* is represented as a *ParameterBinding* (the *IFML ParameterBindingGroup*). The values of the *Parameter* `Title` and `Year`, which denote the search values entered by the user in the `AlbumSearch` *ViewComponent*, is associated with the value of the input *Parameters* `AlbumTitle` and `AlbumYear` which is requested for the computation of the `AlbumList` *ViewComponent*. The `AlbumList` *ViewComponent* specifies a list for the search results where the user can select one specific album item that is triggered through the `SelectItemEvent`. The `SelectedAlbum` is passed through a *ParameterBinding* to the target *ViewComponent* `AlbumDetails` which is responsible for showing detailed information about the selected album item. The tree view representation for the above described IFML model example is shown in Figure 2.11.



```
IFML Model
  Interaction Flow Model
    IFML Window AlbumSearch
      Form AlbumSearch
        IFML Parameter Title
        IFML Parameter Year
        On Submit Event OnSubmitSearch
          Navigation Flow _oop-IAI8Eeq_mf8mlrc4cA
    IFML Window Albums
      List AlbumList
        IFML Parameter AlbumTitle
        IFML Parameter AlbumYear
        IFML Parameter Title
        IFML Parameter Year
        On Select Event OnSelectAlbumEvent
          Navigation Flow _yKQQMAI8Eeq_mf8mlrc4cA
            Parameter Binding Group _sKKgkAI9Eeq_mf8mlrc4cA
              Parameter Binding _tLk6QAI9Eeq_mf8mlrc4cA
              Parameter Binding _tiHyAAI9Eeq_mf8mlrc4cA
    IFML Window Album
      Details AlbumDetails
        IFML Parameter AlbumTitle
        IFML Parameter AlbumYear
  Domain Model
```

Fig. 2.11 Tree view representation of the *IFML* model example

In summary, IFML is an OMG standardized UI modeling language for describing the content, user interaction and control behavior of the front-end of software applications. Although it serves as a solid solution for UI modeling, it lacks means for explicitly modeling aspects such as context management and UI adaptation which have to be taken into account when it comes to the development of self-adaptive UIs.

2.3 Context-Aware Computing

As context management aspects (e.g., context monitoring, change detection, etc.) represent an important prerequisite for realizing self-adaptive UIs, the following Section introduces the term *Context-aware Computing* and its main concepts.

Context-aware Computing stems from the vision articulated by Marc Weiser in his seminal paper: 'The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it' [Wei99]. The goal of context-aware computing is to acquire and utilize information pertaining to the physical world, and then select, configure, and provide a variety of services accordingly.

Context-aware systems are concerned with the acquisition of context (e.g., using sensors to perceive a situation), the abstraction and understanding of context (e.g., matching a perceived sensory stimulus to a context), and application behavior based on the recognized context (e.g., triggering actions based on context). Context-awareness is regarded as an enabling technology for ubiquitous computing systems.

In the following, we introduce the notion of context based on [Sch13] and describe relevant information regarding context acquisition through sensors and modeling context-of-use situations.

2.3.1 Notion of Context

For the creation of flexible and highly usable user interfaces it is essential to understand the context-of-use. With context-aware computing, we have the means of considering the context-of-use not only in the design process, but also at runtime while the device is in use. In Human-Computer Interaction (HCI), it is important to understand the user and the context-of-use and create designs that support the major anticipated use cases and situations of use. In Context-Aware Computing, on the other hand, the consideration of context causes a fundamental change: We can support multiple contexts-of-use that are equally optimal. At runtime, when the user interacts with the application, the system can decide what the current context-of-use is and provide a user interface specifically optimized for this context.

With context-awareness, the job of designing the user interface typically becomes more complex as the number of situations and contexts which the system will be used in usually increases. In contrast to traditional systems, we do not design for a single or a limited set of contexts-of-use. Instead, the overall goal in context-aware computing is to design for several contexts. The advantage of this approach is that we can provide optimized user interfaces for a range of contexts [Sch13].

In the early days of the computing era, the context in which systems were used was strongly defined by the place in which computers were set up. Personal computers were used in office environments or on factory floors. The context-of-use did not change much, and there was little variance in the situations surrounding the computer. Hence, there was no need to adapt to different environments. Many traditional methods in the discipline of Human-Computer Interaction (HCI), such as contextual inquiry or task analysis, have their origin in this period and are most easy to use in situations that do not constantly change. With the rise of mobile computers and ubiquitous computing, this changed [Sch13]. Users take computers and smart devices with them and use them in many different situations.

The term context is widely used with very different meanings. A rather generic description of what constitutes context, has been provided by Dey [Dey01]:

> “*Context* describes any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.‘’

For defining the term context, the research community on model-based user interfaces shift the emphasis towards the notion of context-of-use. Much of the work in this area builds on the unifying reference framework by Calvary et al. [CCT+03], who define the context-of-use of an interactive system by three classes of entities:

- The *users* of the system who are intended to use (and/or who effectively use) the system,
- the hardware and software *platform*(s), that is, the computational and interaction device(s) that can be used (and/or are used, effectively) for interacting with the system, and
- the physical *environment* where the interaction can take place (and/or takes place in practice).

Under software engineering considerations, Calvary et al. [CCT+03] distinguish between predictive contexts-of-use that are known at design-time and the effective contexts-of-use that can only be determined at runtime, and which may differ.

2.3.2 Context Acquisition and Modeling

An important prerequisite for context-aware computing is the acquisition of contextual information. It can have different sources, although the most relevant are sensors in our environment. In that case we can refer to information as “sensed context information‘‘. These sensors can have various forms and deliver very different types of results. All of them are object to failure and noise. There is usually a significant gap between sensor output and the level of information that is useful to applications. Therefore, this gap must be bridged by various kinds of processing of context information before the information is passed to context-aware services [HS09].

Most current devices, by which web and mobile applications are accessed today, have a diverse set of sensors to support a broad spectrum of applications. Those are primarily intended to be utilized by the operating system or native applications. But browsers continuously support the access of hardware sensors via APIs or libraries. Standards for sensor access are set by the World Wide Web Consortium (W3C)[3] and implemented by different browsers, such as Google Chrome or Mozilla Firefox. Not all standards are implemented right away or even get reverted, e.g., for privacy concerns.

A specific property of context may be derived by different sources. Like the current position can be derived from the IP address or, if existing, from a GPS sensor. Moreover, different sources of sensory information can be combined to derive new properties. For example, the accelerometer and the light sensor combined with even more sensors are being used to detect if a smartphone is inside a pocket or in the hand of a user. This enhances the contextual information, but that is not the main focus of this thesis. It will be assumed that the needed information for the definition of UI adaptations can be obtained primarily through the use of external APIs and libraries, which themselves can access diverse sensory information and therefore upvalue the given contextual information themselves. For example, the AffectivaSDK[4], utilized in this thesis, accesses the front-camera of the used device and derives the current mood of the user. This means that the SDK enhances pure information

[3]https://www.w3.org
[4]https://developer.affectiva.com

about the view in front of the device to information about the current state of the user.

According to [GS01], sensed context is information and can therefore be modeled as such. It contains important characteristics that are utilized in a model:

- information content
  - sensed properties
  - the subject of sensing
- meta information
  - information quality attributes
  - information about the source of the content

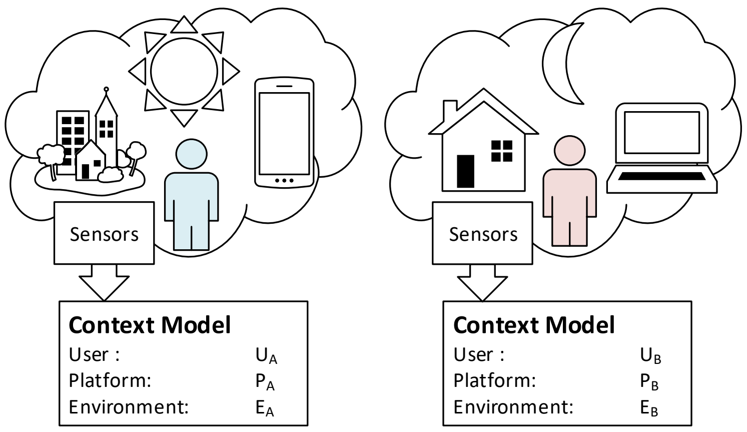


Fig. 2.12 Abstracting context-of-use with a context model

As explained in [dK01], a model is a simplification of reality and abstracts systems' concepts, including their properties, methods, relationships, cardinalities, and constraints. This shall provide a version of a system sufficiently complete to comprehend the systems' goals. That facilitates a possibility to describe the context of the regarding application. Different circumstances will lead to a different state of the context model, as shown in Figure 2.12. A metamodel serves as a basis for developers to extend existing models or create new ones. Different approaches for modeling context have been developed over time. In general, the approaches can be classified according to the following types, as stated in [SLP04] (for every type, an example work is given):

- *Key-Value Models*: Represent simple structure for context information. Services are described by a list of simple attributes in a key-value manner. They are easy to manage, but lack capabilities for structuring for efficient context retrieval [SAW94].
- *Markup Scheme Models*: Hierarchical data structure consisting of markup tags with attributes and content. Content is usually recursively defined by other markup tags. Languages are often based on generic markup languages like XML [HBSS02].
- *Graphical models*: Often based on general purpose modeling artifacts such as *Unified Modeling Language* (UML) diagrams. Standard tools are often extended to enlarge the context modeling capabilities [HIR03].
- *Object-Oriented Models*: This is based on the concept of objects and relation-ships between them as in the object-oriented programming paradigm. Object-oriented context models aim to utilize encapsulation and increase reusability. Access to information is provided through specified interfaces only [SBG99].
- *Logic-Based Models*: Facts, expressions, and rules are used to define a context model. The logic defines the conditions on which a concluding expression or fact is derived from a set of other expressions or facts. These models have a high degree of formality [GS01].
- *Ontology-Based Models*: This approach is based on ontology models and enables to specify concepts and interrelations of information. Some also allow for consistency checking and contextual reasoning [zA97].

Above listed context modeling approaches have different capabilities and are suitable for different problems. The application of a concrete context modeling approach depends on the specific requirements of the application scenario and domain.

2.4 Self-adaptive Software Systems

In this section, we provide basic information about self-adaptive software systems by describing the background and concept of autonomic computing. Then, we explain self-adaptive software systems and different types of self-* properties in general. After introducing self-adaptive software systems in general, we transfer the idea of self-adaptivity to the context of user interfaces. For this purpose, we present different types of UI adaptation techniques and differentiate between different self-* properties of self-adaptive UIs.

2.4.1 Background: Autonomic Computing

The term *Autonomic Computing* (also known as AC) refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users [KC03]. Autonomic Computing, initiated by IBM in 2001, helps to address the development of computer systems capable of self-management, to overcome the rapidly growing complexity of computing systems management, and to reduce the barrier that complexity poses to further growth [IBM05]. The term autonomic is derived from human biology. The autonomic nervous system monitors the heartbeat, checks the blood sugar level, and keeps the body temperature in an ideal degree without any conscious effort needed from the human side. In much the same way, self-managing autonomic capabilities anticipate IT system requirements and resolve problems with minimal human intervention [IBM05].

The AC system concept is designed to make adaptive decisions, using high-level policies. It will constantly check and optimize its status and automatically adapt itself to changing conditions. The MAPE-K loop was introduced by IBM as a reference architecture for autonomic computing [IBM05]. As depicted in Figure 2.13, the reference architecture for autonomic computing integrates the two main components *Autonomic Manager* and *Managed Resource*.

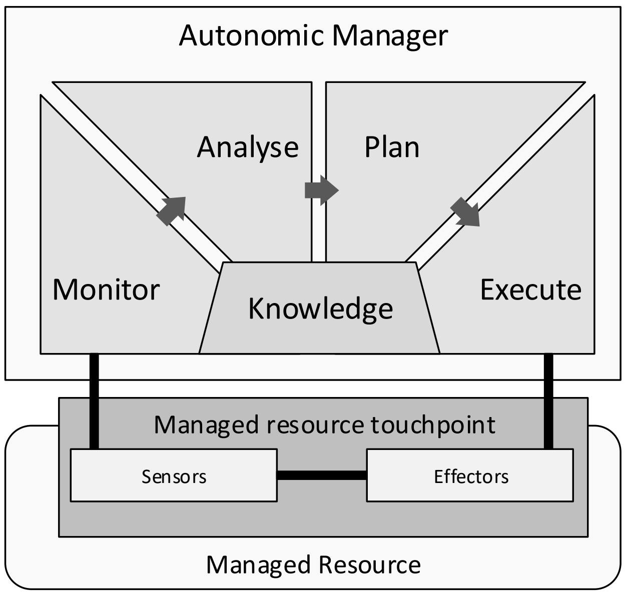


Fig. 2.13 IBM MAPE-K Loop [IBM05]

A *Managed Resource* is a hardware or software component that can be managed. A *Managed Resource* could be a server, storage unit, database, application server, service, application or other entity like the user interface as it is the case in this work. A *touchpoint* is an autonomic computing system building block that implements sensor and effector behavior for one or more of a managed resource's manageability mechanisms. Through a touchpoint the *Managed Resource* provides the ability to be sensed (via *Sensors*) and changed (via *Effectors*) by the *Autonomic Manager*.

The *Autonomic Manager* is responsible for continuously monitoring the *Managed Resource* through *Sensors* and to automatically react to changing conditions by adapting the *Managed Resource* through *Effectors*. For this purpose, the *Autonomic Manager* consists of a control loop that is called MAPE-K, while this acronym represents the starting letters of the main sub components:

- *Monitor*: The monitor component provides the mechanisms that collect, aggregate, filter and report details collected from a managed resource.
- *Analyze*: The analyze component provides the mechanisms that correlate and model complex situations (for example, time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations.
- *Plan*: The plan component provides the mechanisms that construct the actions needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work.
- *Execute*: The execute component provides the mechanisms that control the execution of a plan with considerations for dynamic updates.
- *Knowledge*: Data used by the autonomic manager's four components (monitor, analyze, plan and execute) are stored as shared knowledge. The shared knowledge includes data such as topology information, historical logs, metrics, symptoms, and policies.

2.4.2 Self-adaptation and Self-*properties

Many researchers use the terms *Autonomic Computing* and *Self-adaptive* (not specifically self-adaptive software) interchangeably. Whereas the term of *Autonomic Computing* has emerged in a broader context and covers the complete layers of a software intensive system that consists of application(s), middleware, network, operating system, and hardware, *Self-adaptive*

Software Systems are more specific and primarily cover the application and middleware layers. This means self-adaptive software systems are a specific subclass of autonomic systems and fall under their umbrella [ST09]. A more concrete definition for self-adaptive software, is provided in a DARPA Broad Agency Announcement (BAA) by Laddaga [Lad97]:

> "Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."

A similar definition is given in Oreizy et al. [OGT$^+$99]:

> "Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation."

Both definitions underline the aspect that *Self-adaptive software* aims to adjust various artifacts or attributes in response to changes in the *self* and in the *context* of a software system. By self, the whole body of the software is meant, mostly implemented in several layers, while the context encompasses everything in the operating environment that affects the system's properties and its behavior. Therefore, a self-adaptive software is a closed-loop system with feedback from the self and the context.

Self-adaptive software is expected to fulfill its requirements at runtime in response to changes. To achieve this goal, software should have certain adaptivity characteristics, known as self-* properties ([IBM05], [BJM$^+$05]). The need for self-adaptive software or adaptivity capabilities can have a broad range of reasons. For instance, a self-adaptive software system might react to erroneous system states or optimize its own behavior to perform a specific task more efficiently. Based on the initial well-known set of self-* properties, introduced by IBM, Salehie et al. [ST09] present a hierarchical set of self-* properties (see Figure 2.14) which are briefly explained in the following.

General Level: This level contains global properties of self-adaptive software. A subset of these properties, which falls under the umbrella of self-adaptiveness, consists of self-managing, self-governing, self-maintenance, self-control and self-evaluating.

Major Level: The IBM autonomic computing initiative defines a set of four properties at this level. This classification serves as the de facto standard in this domain. These properties

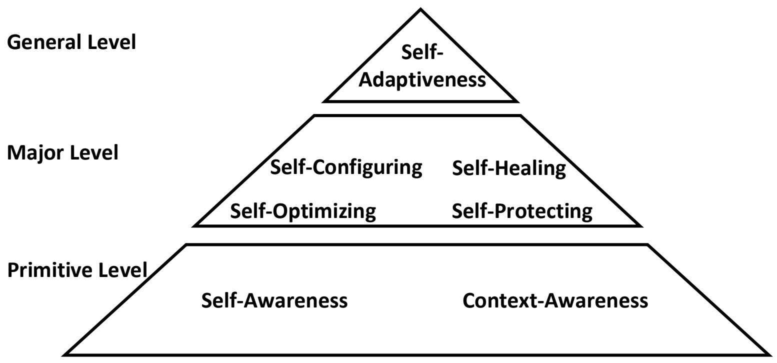


Fig. 2.14 Hierarchy of the self-* properties [ST09]

have been defined in accordance to biological self-adaptation mechanisms. The following list further elaborates on the details.

- *Self-configuring* is the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing/decomposing software entities.
- *Self-healing*, which is linked to self-diagnosing or self-repairing, is the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and accordingly take proper actions to prevent a failure. Self-diagnosing refers to diagnosing errors, faults, and failures, while self-repairing focuses on recovery from them.
- *Self-optimizing*, which is also called self-tuning or self-adjusting, is the capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to-end response time, throughput, utilization, and workload are examples of important concerns related to this property.
- *Self-protecting* is the capability of detecting security breaches and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating problems and taking actions to avoid them or to mitigate their effects.

Primitive Level: Self-awareness and context-awareness are the underlying primitive properties. The following list further elaborates on the details.

- *Self-Awareness* means that the system is aware of its self states and behaviors. This property is based on self-monitoring which reflects what is monitored.
- *Context-Awareness* means that the system is aware of its context, which is its operational environment.

2.4.3 Self-adaptive User Interfaces

The user interface (UI) layer is considered as one of the key components of software applications since it connects their end-users to the functionality. Providing an easy to use and adequate user interface that is suitable for the current contextual situation of the end-user is a complex task due to dynamically changing heterogeneous context-of-use situations. As discussed in the previous subsections, self-adaptive software systems cover the application layer and thus self-* properties can be also integrated in the UI layer to increase usability and flexibility of the user interface.

Based on the described hierarchy of self-* properties for self-adaptive software systems by Salehie et al. [ST09] (see Figure 2.14), Akiki et al. [ABY14a] identify self-* properties for self-adaptive UIs. In their literature study, they noticed that some of the problems are technical and related to devising systems that can support the development of self-adaptive UIs, while others are related to human factors such as the end-user acceptance of these UIs. As depicted in Figure 2.15, the authors point out the essential self-* properties which should be realized to handle the relevant technical and human problems related to self-adaptive UIs.

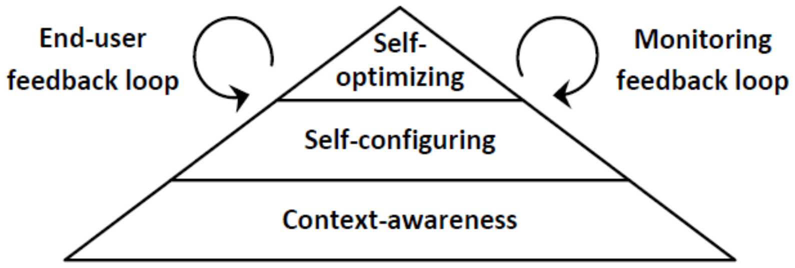


Fig. 2.15 Self-* properties of Self-adaptive UIs [ABY14a]

In the following, based on [ABY14a], the main adaptivity characteristics relevant to the domain of self-adaptive UIs are described (see Figure 2.15).

- *Context-awareness*: 'indicates that a system is aware of its context, which is its operating environment'. If the UI is aware of its context and is able to detect context changes, then it can trigger adaptations (e.g., based on a set of rules) in response to those changes in order to preserve its usability.
- *Self-configuring*: 'is the capability of reconfiguring automatically and dynamically in response to changes'. To keep the UI adaptation rules up to date with an evolving

context-of-use (e.g., if a user's computer skills improve), there is a need for a mechanism that can reconfigure these rules by monitoring such changes. Another type of rule reconfiguration could be based on the end-users' feedback. For example, the end-user may choose to reverse a UI adaptation or select an alternative. Keeping the end-users involved in the adaptation process could help in increasing their awareness and control, thereby improving their acceptance of the system.

- *Self-optimizing*: 'is the capability of managing performance and resource allocation in order to satisfy the requirements of different users'. To adapt this definition to user interfaces, we can say that a UI can self-optimize by adapting some of its properties. For example, adding or removing features, changing layout properties (e.g., size, location, type, etc.), providing new navigation help, etc.

Beside the above described self-* properties for self-adaptive UIs, it is also important to consider the different forms and techniques for UI adaptation. A systematic classification for different types of possible adaptations in the context of user interfaces is provided by Nebeling [Neb12], who differentiates between the following main types of UI adaptations:

- *Adaptation of Content / Task-feature set.* According to Brusilovsky [23], content-level adaptation techniques include conditional text or stretch text to expand and collapse foldable text paragraphs. Other typical examples of content-level adaptation include internationalization and localization of web sites as these primarily require the translation of content, but may also concern the reading order and thus the alignment of page elements. With regard to content adaptation, it is also important to mention that task-feature set adaptations enable to adapt the amount of shown display and interaction elements. Based on the current context-of-use the task-set of the UI can be increased or decreased to better support the need of the users.

- *Adaptation of Navigation.* Navigation-level adaptations may support direct guidance of a user to relevant content within the current document or to related pages. The explored techniques include structural or link-level adaptations such as hyperlink sorting, hiding and annotation techniques [23]. The second survey by Brusilovsky [24] further distinguishes the techniques used for link hiding and also discusses those for generating new links. Link hiding can be achieved by temporarily or permanently removing hyperlinks or by disabling them. Link generation techniques may be used for dynamically adding a list of relevant links for the current document or new useful links between related pages, as well as creating links based on similarity between navigation items.

- *Adaptation of Presentation.* Presentation-level adaptations concern the design, i.e. the font, color and/or style, or the layout, i.e. the position, size and/or order (e.g., z-index) of rendered page elements. While Brusilovsky [23, 24] classified many different works into using either adaptive text presentation or adaptive navigation techniques, adaptive layout techniques often use a combination of different adaptation technologies that are difficult to tell apart and thus to classify. For example, techniques such as stretch text are special since they are concerned with both changing the text and the layout when additional content associated with keywords is expanded or collapsed within the document and, as a result, shifts other elements in the page [162]. However, some techniques such as text dimming that Brusilovsky [24] characterised as adaptive text may be better labeled as an adaptive layout technique, as they essentially only change the presentation of content and not the content as such.

- *Adaptation of Modality.* Beside the above introduced types of UI adaptation techniques, UI changes regarding the interaction modality can be also seen as a further adaptation technique as illustrated in [Pat13]. With this regard, it is possible to provide different interaction modalities for the end-users. Based on the current context-of-use one can switch for between different modalities (e.g., textual UI, graphical UI, vocal UI, etc.

2.5 Usability Engineering

In this section, we give a brief overview on the topics usability and usability evaluation methods as usability evaluation is an essential part of our model-driven engineering approach for self-adaptive UIs. First of all, we introduce the main ideas behind usability and relevant usability criteria that should be considered in the development process of interactive systems. After that, we briefly present existing usability evaluation methods that are used for assessing the quality of use of interactive systems.

2.5.1 Usability

Usability is the ease of use and learnability of a human-made object such as a tool or device [Sta98]. In software engineering, usability is the degree to which a software can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context-of-use. According to ISO 9421-11 [Sta98] usability focuses on the main criteria of executing tasks with effectiveness, efficiency, and satisfaction. Effectiveness means that the tasks are fully completed. Efficiency refers to the effort for task execution which should be as low as possible. Satisfaction, in addition, considers that the task execution must be pleasant for the users. As the definition already shows, usability depends on the usage context, which covers user groups, tasks to be executed, as well as the physical and social environment of the user. Therefore, usability may vary strongly between different usage contexts which means, e.g., that a product can have a high usability for one person and a low usability for another.

As described in [HG14], a usability issue is a problem with the software that decreases its usability. This means, it decreases one or several factors of effectiveness, efficiency, and satisfaction. Usability issues can have different causes like the visual design, the information architecture, the performance, or failures of a software. For example, a specific color combination in the visual design can make it hard for users to identify a certain GUI element and, therefore, to fulfill a task (effectiveness). Furthermore, the information architecture may require users to perform long navigation paths through a website (efficiency) to reach a specific information.

Beside the ISO definition, a further widely accepted notion for usability comes from usability consultant Jakob Nielsen [Nie93] and computer science professor Ben Shneiderman [Shn00] who have written (separately) about a framework of system acceptability, where usability is a part of 'usefulness' and is composed of:

- *Learnability:* How easy is it for users to accomplish basic tasks the first time they encounter the design?
- *Efficiency:* Once users have learned the design, how quickly can they perform tasks?
- *Memorability:* When users return to the design after a period of not using it, how easily can they re-establish proficiency?
- *Errors:* How many errors do users make, how severe are these errors, and how easily can they recover from the errors?
- *Satisfaction:* How pleasant is it to use the design?

2.5.2 Usability Evaluation Methods

The major goal of a usability evaluation is to measure different aspects of the usability of a software, like effectiveness, efficiency or satisfaction. Basically, usability evaluation aims at identifying usability issues. This requires the predefinition of evaluation goals, the analysis of the usage context, and finally the measurement and assessment of usability aspects using dedicated methods. The analysis of the usage context includes the identification of typical tasks users perform with a software. These tasks serve as input for the evaluation methods.

There are a variety of usability evaluation methods. Certain methods use data from users, while others rely on usability experts. There are usability evaluation methods for all stages of design and development, from product definition to final design modifications. As described in [Har16], usability evaluation methods can be subdivided into expert- and user-oriented methods.

Expert-oriented methods are performed by experts who know how a specific method must be applied. These methods define concrete steps the expert has to take to identify usability issues. For example, an expert measures the achievable efficiency of a user executing a specific task by identifying detailed actions a user has to take and then estimating the average time for the action executions.

In contrast, user-oriented methods follow a process in which users use a prototype or a running software for predefined tasks while they are observed by an evaluator. The observations are then analyzed and help to identify usability issues. For gathering data during the observations, different methods like taking notes, recording user actions, letting users fill out questionnaires, or thinking aloud can be applied. User-oriented usability evaluation or usability tests can be done in a laboratory or in the field [KSR03]. A laboratory setup might influence the user and, hence, the evaluation results. When done in the field, user-oriented usability evaluation lets users do their tasks in their natural environment, i.e., in the matching usage context making the results more reliable [KSR03].

As a further usability evaluation method it is important to mention model-based usability evaluation. Model-based usability evaluation methods utilize models (e.g., user models, UI models, etc.) which can describe users and the way they utilize a software [AIV08] or the software itself [Træ02]. For example, a model can define average durations for specific actions or it may describe the graphical UI. Usually, the model is created before and analyzed during the evaluation. A model can be created manually or automatically where in the latter case it is usually derived from other models (like the GUI itself) through model transformation.

Another important term related to usability is usability engineering. Usability engineering describes the continuous application of usability evaluation methods during the development process of a software with the goal to achieve high usability of the final product [Nie93]. The application of the evaluation methods requires preparation. Therefore, usability engineering usually covers five tasks: a) analysis of users and context, b) modeling of a solution, c) specification of solution details, d) realization, and e) evaluation of the solution. These tasks must not be understood as successive but as contributing to each other. For example, a modeling may result in requiring a further analysis of a specific aspect. Usability evaluation methods are applied in Task e) but require preparation in all other tasks.

It is important to distinguish between usability testing and usability engineering. Usability testing is a specific activity as part of usability engineering to measure the ease of use of a product or piece of software. In contrast, usability engineering (UE) is broader and contains the research and design process that ensures a product with good usability. Usability is a non-functional requirement. As with other non-functional requirements, usability cannot be directly measured but must be quantified by means of indirect measures or attributes such as, for example, the number of reported problems with ease-of-use of a system.

Related to usability are the terms user experience and accessibility. User experience covers a broader context than usability [LP08]. In addition to usability, it considers an '... individual's entire interaction with the [software], as well as the thoughts, feelings, and perceptions that result from that interaction.' [TA13]. In some definitions, user experience covers a whole customer journey from searching for a product, via buying it, up to using support during usage. Usability can be seen as an important part contributing to the user experience. In comparison to the before mentioned terms, accessibility aims at making software usable for people with certain disabilities. Self-adaptive UIs as aimed in this thesis can also support improving accessibility issues.

2.6 Technologies

In this section, we give a brief overview of the relevant technologies that were used to implement the solution approach of this thesis. Therefore, we introduce the *Angular* framework, the *Nools* rule engine, and describe basic concepts behind *Xtext* and *Xtend*.

2.6.1 Angular Framework

The Angular framework[5] is one of the most used UI/Web frameworks. Angular is maintained by Google and aims to facilitate the development of modern UIs and (web) applications for different target platforms by relying on established development practices, concepts, and conventions. Angular has established itself as a de facto standard for front-end/UI development purposes and is highly used in industrial projects. Angular is based on the languages JavaScript[6] and HTML[7]. Usually, TypeScript[8], a strict syntactical superset of JavaScript is used while developing Angular applications. It provides a flexible and easy to use module system as well as type system to ease the work of the developers. As illustrated in Figure 6.2, the architecture of the Angular framework is very modular.

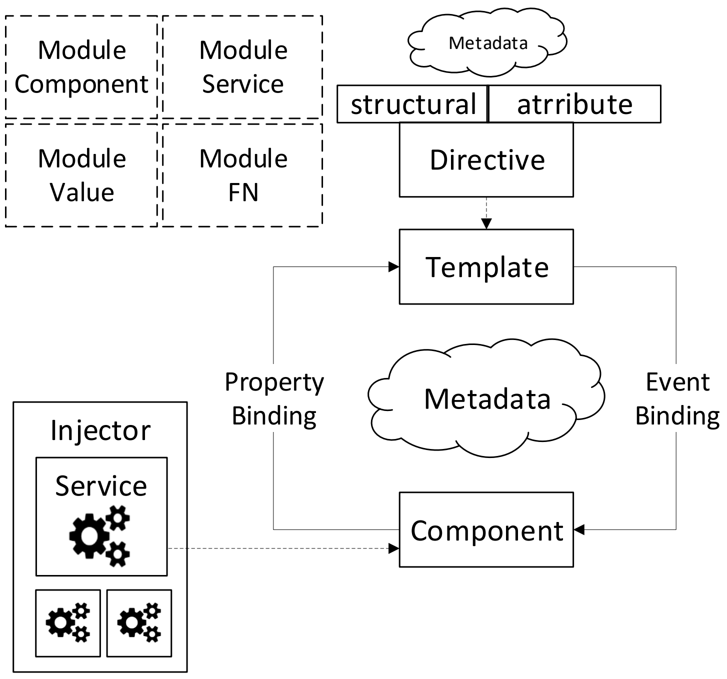


Fig. 2.16 Architectural Overview of the Angular framework [GS13]

In the following, the individual building blocks of the Angular framework are shortly explained.

[5]https://angular.io
[6]https://www.javascript.com/
[7]https://www.w3.org/html
[8]https://www.typescriptlang.org

An Angular application is modular and assembled from multiple *Modules*. *Modules* are a collection or grouping of code artifacts which support a common goal. Although they are optional, use of *Modules* is encouraged to improve software quality (e.g., readability, maintenance, etc.). *Modules* can be made public and imported by use of the statements *import* and *export* respectively.

A *Component* controls parts of the view, e.g. a list which is displayed. The *Component*'s application logic is defined within a class, which can interact with the view. Angular autonomously manages (creates, updates, and destroys) the *Components* as they are needed.

The view with which a *Component* interacts is called *Template*. A *Template* is represented as HTML code and specifies how the *Component* is displayed. However, additionally to the HTML code, a *Template* also contains Angular markup code. This includes, for example, other *Components* or different forms of data binding. Data binding is a mechanism that connects the *Component* and the *Template*. The different kinds of data binding allow to establish a connection in both directions, also called two-way data binding, or just in one direction (from *Component* to *Template*, also called *Property Binding*, or the other way round, also called *Event Binding*).

Angular *Templates* are rendered dynamically according to the instructions given by the *Directive*. *Directives* can be categorized into two groups: *structural* and *attribute Directives*. Structural directives can be used for broader layout changes. They can add, remove or replace elements in the Document Object Model (DOM), which is an interface for accessing the HTML structure. Attribute directives, on the other hand, enable to change the appearance or behavior of an element.

Services in Angular enable to integrate application functionality into *Components* through an *Injector*. Simple examples for services could be logging, user input validation, data fetching or calculation functions. Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity. Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself. In Angular, the DI framework provides declared dependencies to a class when that class is instantiated.

As described above, the architecture of the Angular framework fits well to to our concern to characterize the UI and reflect UI changes to realize runtime UI adaptation. Moreover, the architecture of the Angular framework is modular and supports sufficient flexibility to integrate further aspects such as context monitoring and UI adaptation at runtime.

2.6.2 Nools Rule Engine

A rule engine, in general, is a software component, which having some knowledge based on facts, is able to perform conclusions by executing one or more rules in a runtime environment. Commonly known rule engines are business rule engines from the domain of enterprise applications. In this domain, a rule engine is used to separate business rules (e.g., "‘*All customers that spend more than 100€ at one time will receive a 10% discount*"’) from the application code. The usage of a rule engine brings many advantages which are described in the following:

- *Understandability:* Rules are easier to understand for a business analyst or a new developer than a program written in Java or other imperative-style language.
- *Maintainability:* Since rules are specified in a declarative manner and separated from application logic, a developer can spend more time solving the actual problem in his or her domain of expertise.
- *Flexibility:* Rule engines bring flexibility to better cope with changes to the requirements or changes to the data model. Changing or rewriting an application is never an easy task. However, thanks to the formalism that rules bring, it is much easier to change rules than to change the application logic.
- *Efficiency and Scalability:* Here the advantage comes from the existing efficient algorithms such as the RETE [For82] algorithm which perform efficiently, accurately and quickly .
- *Reusability:* The rules are kept in one place and are separated from the application logic. This way, already specified beneficial rules can be used in other contexts (e.g., for another application, domain, project etc.).
- *Runtime Continuity:* It is possible to change/redeploy rules and processes without even stopping the whole application. That means, while the rule engine is executed, the set of rules can be changed at runtime.

In our solution for model-driven development of self-adaptive UIs, we aim to reduce the complexity for dealing with UI adaptations by introducing a rule engine for managing UI adaptations. As our approach focuses on web applications as target technology, we decided to use the *Nools* rule engine for managing the UI adaptation rules and enabling runtime UI adaptation behavior. *Nools* is an efficient rule engine which is based on the *RETE* algorithm, developed by Charles L. Forgy [For82]. It is based on JavaScript and suits well in the context of web application development as it is compatible and can be easily integrated into existing web frameworks such as *Angular*.

As illustrated in Figure 2.17, *Nools* can be used to evaluate a set of rules at runtime. *Rules* in *Nools* are organized in form of a *Flow*. A *Flow* acts as a container for *Rules*. The evaluation of *Rules* is done with a so called *Session*, which is an instantiation of a *Nools Flow*. The *Session* can be obtained directly from the *Flow* object. To start a *Session*, first *Facts*, that should be asserted, need to be set. A *Fact* is an item which is checked for matches against the conditions defined in the *Rules*.

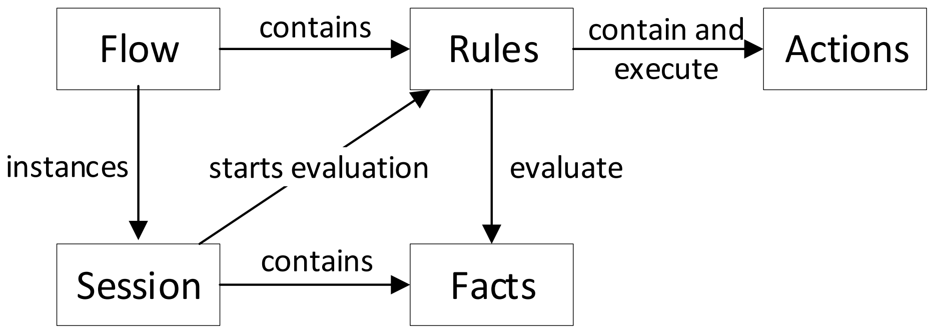


Fig. 2.17 Overview of *Nools*' basic concepts

A *Rule* has several non-optional parts. First part is a unique *Rule Name* for identifying the *Rules*. Secondly, a rule *Rule* conditions that should be evaluated. And lastly, a sequence of *Actions* that should be performed. The *conditions* are an array of either a single condition or a sequence of conditions. A list of available operators for the conditional expressions can be seen in the *Nools* documentation[9]. *Actions* are functions that are fired if all conditions of the rule are satisfied. They are called within the scope of the rule engine and get passed the facts as an argument. Optional parameters of the rule include options to prioritize the execution of different rules. The priority option is called *salience* and is given by a number. The prioritization is done by the developer and to some extent allows conflict resolution between rule executions. Rules with higher salience are executed before rules with lower salience.

In Code Excerpt 2.1, a small example for a simple *Nools Flow* is shown. First, *Nools* is loaded as required module and the *Fact*, `Message`, is defined. The *Flow* `Hello World`

[9]http://c2fo.io/nools/#constraints

contains two rules taking a message as a *Fact*. *Rule* `Hello` is fired if the text message starts with `hello world`. Then, in the *Action* the text message is concatenated by `goodbye` and the *Fact* is modified with the changed text. Now, the condition for the *rule* `Goodbye`, ending with the text message `goodbye`, is satisfied and the corresponding *Action* is fired. The *Action* prints the text message to the JavaScript console.

```javascript
var nools = require("nools");
var Message = function (message) {
    this.text = message;
};
var flow = nools.flow("Hello World", function (flow) {
    //find any message that start with hello
    flow.rule("Hello", [Message, "m", "m.text =~ /^hello\\sworld$/"],
        function (facts) {
        facts.m.text = facts.m.text + " goodbye";
        this.modify(facts.m);
    });
    //find all messages then end in goodbye
    flow.rule("Goodbye", [Message, "m", "m.text =~ /.*goodbye$/"],
       function (facts) {
        console.log(facts.m.text);
    });
});
```

Code Excerpt 2.1 Nools rule example

To sum up, the **Nools** rule engine has been used to realize context monitoring and UI adaptation for self-adaptive UIs. Its concrete role and application in our solution will be explained in the upcoming chapters.

2.6.3 Xtext and Xtend

As one of the main goal of this thesis is to support model-driven development of self-adaptive UIs, the development of domain-specific languages and code generation are essential parts of this work. For addressing these aspects on an implementation specific level, two common technologies exist: Xtext[10] which supports the development of programming languages and domain-specific languages, and Xtend[11] which enables flexible support for realizing template-based code-generators. In the following, both technologies are briefly described.

[10]https://www.eclipse.org/Xtext
[11]https://www.eclipse.org/xtend

Xtext

Xtext is a framework for development of programming languages and domain specific languages. Xtext provides a flexible grammar language to define various languages. In our case, Xtext is used to define modeling languages to cover the context management and UI adaptation concerns. The main advantage of Xtext is that it provides a full infrastructure, including parser, linker, typechecker, compiler as well as editing support for Eclipse or any editor that supports the Language Server Protocol. Xtext's language parser creates an abstract syntax tree from the code, which is easily traversable by a generator, which can be implemented with Xtend (see next page). In the following, the main idea of Xtext is shown based on an example grammar.

```
grammar org.example.domainmodel.Domainmodel

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

    (elements+=PackageDeclaration)*;

PackageDeclaration:
    'package' name=QualifiedName '{'
        (elements+=Entity)*
    '}';

QualifiedName:
    ID ('.' ID)*;

Entity:
    'entity' name=ID '{'
        (features+=Feature)*
    '}';

Feature:
    (many?='many')? name=ID ':' type=QualifiedName;
```

Code Excerpt 2.2 Xtext example grammar

In Code Excerpt 2.2, a small grammar is shown. The root element of the grammar is the *Domainmodel*. It can contain zero or more, indicated by the asterisk, *PackageDeclarations*. Package declarations consist of text in form of 'package' and a name. Within the brackets ('{' and '}') a package can contain *Entities*. Entities are similar to Package declarations, but can contain *Features*. A Feature starts with an optional, indicated by the '?', 'many'-tag and then a name and type declaration for the feature. *ID* is a terminal and can be any user input. Terminals cannot contain any other elements. An implementation of the grammar looks like shown in Code Excerpt 2.3.

```
package pkg.example {

  entity Thesis {
    many chapters: Chapter
  }

  entity Chapter {
    title: String
    content: String
  }

}
```

Code Excerpt 2.3 Xtext example grammar implementation

Xtend

In the following, the Xtend[12] programming language is introduced. Xtend is a statically typed Java dialect providing additional features like type inference and extension methods. Due to the use of the Java type system, Java and Xtend are fully interoperable. Xtend is a suitable technology for supporting the implementation of code generators as it offers multi-line template expressions. This means that Model-to-Text Transformations (M2T) can be flexibly implemented based on this technology. Since Xtend is very close to Java in most aspects, in this section the focus will be on the features extension methods and multi-line template expressions, which are also used in for the implementation of the generators provided in this thesis.

Extension methods allow the modification of existing types by adding new methods to them. Methods can be called by using their first argument as receiver. An example for this can be seen in Code Excerpt 2.4. Both calls of the *doSomething*-method are equivalent. However, the goal of extension methods is to achieve better readable code. This is especially achieved, when method calls can be chained by extensions, instead of being nested.

When writing a code generator, there have to be support of two basic concepts in the programming language. The first task is traversing the model, which can be solved in Xtend and Java equally good. The second task is the creation of templates with strings, which is

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

rather tedious with Java as the string has to be broken up into several parts and concatenated if variables have to be used. Additionally, if new lines or tabs are needed, special characters have to be used for formatting. Xtend offers the use of multi-line template expressions for this.

```
    Entity e = new Entity();

    def static void main(String[] args){

        // passing argument inside parentheses
        doSomething(e)

        // with extension method
        e.doSomething()

    }

    def doSomething(Entity input){
        // method body
    }
```

Code Excerpt 2.4 Example for an extension method

In Code Excerpt 2.5, an example for a template expression is pictured. Variable parts of the template are inserted by using guillemets ($\ll \gg$). Within the guillemets any expression can be specified or any method can be invoked. It is also possible to insert conditional expressions (IF) and loops (FOR). Also, line breaks and indentations are automatically handled. In Code Excerpt 2.6, an example output of a template expression, as illustrated before, can be seen.

```
    def buildString(Package p) '''
        <package>
         «FOR e : p.entities»
                <entity>
                    «FOR a : e.attributes»
                        <«a.name» «IF a.isMany == true»many='true'«
                            ENDIF»">
                        </«a.name»>
                    «ENDFOR»
                </entity>
            «ENDFOR»
        </package>
    '''
```

Code Excerpt 2.5 Example for a template expression

```
<package>
  <thesis>
    <chapters many='true'>
    </chapters>
  </thesis>
  <chapter>
    <title>
    </title>
    <content>
    </content>
  </chapter>
</package>
```

Code Excerpt 2.6 Example for a template expression return value

In summary, *Xtext* and *Xtend* are useful technologies for implementing our model-driven development approach for self-adaptive UIs, as they support the development of DSLs and template-based code generators in a practical and easy way.

Chapter 3

Scenario and Related Work

In this chapter, we give an overview of the related work of this thesis. For this purpose, we first describe a running example in Section 3.1 that originates from a real-world scenario and will be used throughout this thesis. Based on this scenario, in Section 3.2, we derive a set of requirements that a solution concept should fulfill. We identify and classify related work in Section 3.3 and evaluate it against the requirements. The findings of this chapter are summarized in Section 3.4.

3.1 LibSoft - The Running Example

The running example is based on a real-world scenario which is derived from the domain of library management. The scenario deals with a library web application for universities which is called *LibSoft*. *LibSoft* provides core library management functionality like searching, reserving, and borrowing books. *LibSoft*'s UI can be accessed by heterogeneous users and user roles (like student or librarian) through a broad range of networked interaction devices (e.g., smartphones, tablets, terminals etc.) which are used in various environmental contexts (e.g., at home, en route, etc.).

Depending on the situation, users are able to access their library services where, when and how it suits them best. Figure 3.1 illustrates such a self-determined cross-channel book borrowing process example scenario, where the user can begin an interaction using one channel (search and reserve a book on her laptop at home), edit the reservation on her way using a mobile channel, and finalize the book borrowing process at the library via self-check-out terminal or at the staff desk.

In the described example scenario, each channel has its own special context-of-use and eventually the contextual parameters regarding user, platform, and environment can dynamically change. Already a small set of contextual parameters can highly influence the

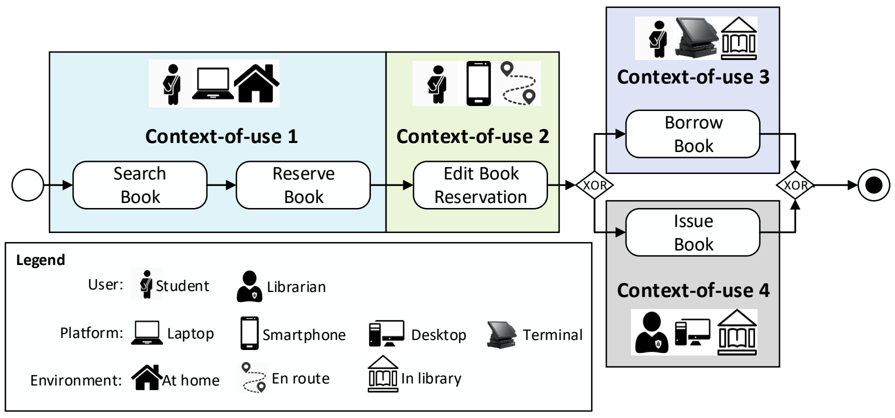


Fig. 3.1 Example scenario: UIs in dynamically changing context-of-use situations

usability of the UI. Therefore, it is important to continuously monitor the context-of-use parameters and react to possible changes by automatically adapting the UI for the new context-of-use situation.

Engineering of such self-adaptive user interfaces is not a trivial task as various aspects have to be taken into account. Regarding the development side of self-adaptive UIs, aspects like context management and UI adaptation additionally increase the complexity and need to be supported in an adequate manner. Also, the usability evaluation of the resulting self-adaptive UI is a challenging task as context and UI are dynamically changing and a suitable way of testing the usability of self-adaptive UIs at runtime is needed. As already shown before, Figure 3.2 recaptures the relevant development and evaluation aspects which are addressed in this thesis. With regards to each of the aspects, we derive in the next section the concrete requirements for the thesis.

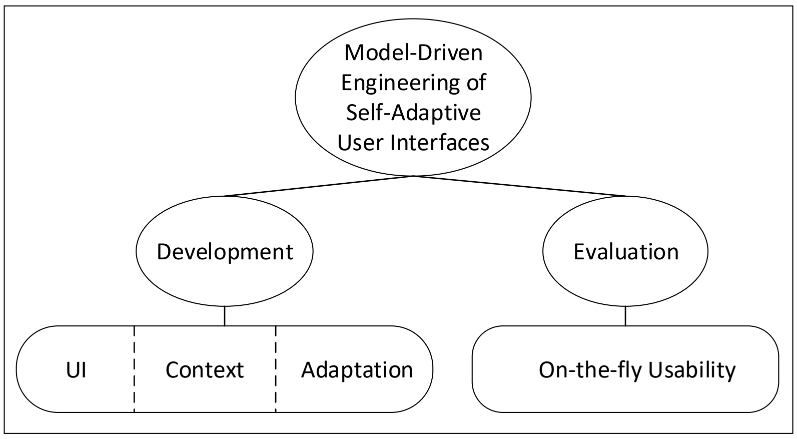


Fig. 3.2 Engineering Self-Adaptive UIs: Overview of relevant aspects

3.2 Requirements

The main goal of this thesis is to support model-driven engineering of self-adaptive UIs. As depicted in Figure 3.3, several requirements need to be fulfilled in order to support the development and evaluation of self-adaptive UIs.

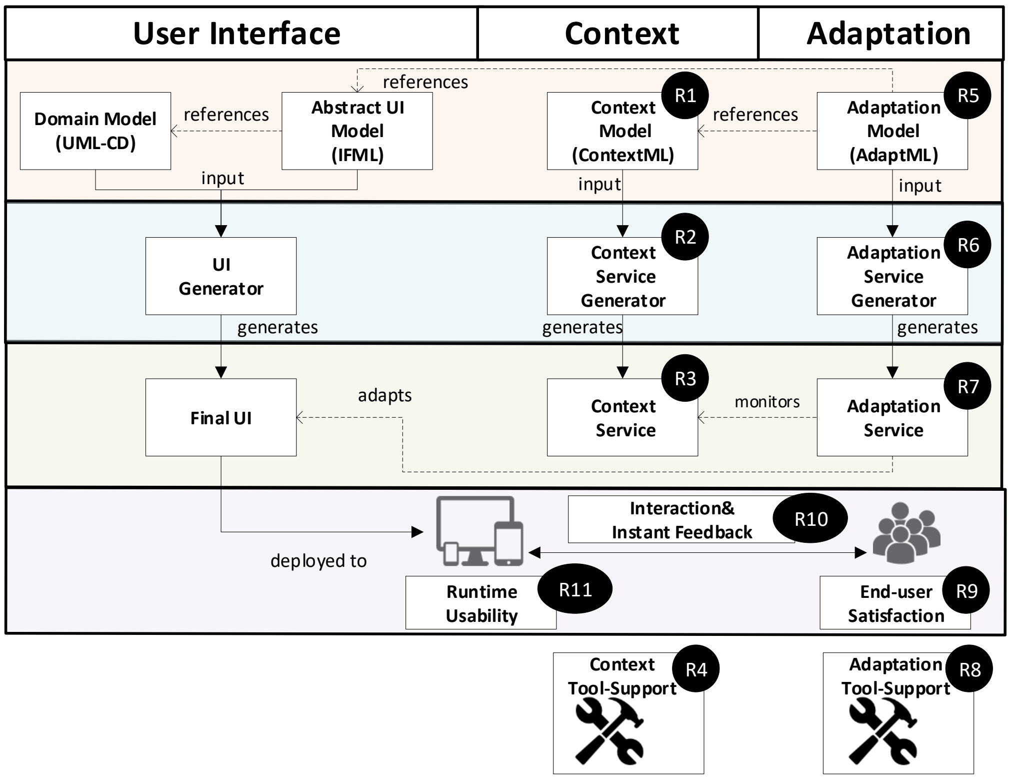


Fig. 3.3 Overview of requirements for development and evaluation of self-adaptive UIs

In the following, based on the relevant aspects *Context Management*, *UI Adaptation*, and *Usability Evaluation* the requirements are described.

Context Management:

- **R1 - Context Modeling Approach**

  R1.1 - Context Modeling: The approach should enable the abstract specification of contextual parameters as an essential prerequisite for UI adaptation. The context model should be integrated in the overall modeling environment of self-adaptive UIs offering a specific viewpoint for modeling context management aspects. The context modeling language should be extensible and it should ease the developer's work in specifying and maintaining contextual parameters.

R1.2 - Context Aspects: The approach should support modeling of various context aspects such as user, platform, and environment characteristics. The context modeling approach should cover various contextual parameters regarding user, platform, and environment in a holistic manner.

- **R2 - 'Context' Transformation Approach**

  The approach should come up with a generative approach where an automated transformation from context models to executable *Context Services* is supported. Based on the specified context model, the developer shall be able to generate code for a *Context Service*. Thus, the amount of repetitive manual code for implementing *Context Services* should be reduced.

- **R3 - Runtime Monitoring**

  The approach should support runtime context monitoring where the generated *Context Services* continuously observe context information and detect context changes through corresponding hardware sensors. The resulting context service shall provide context information to the UI adaptation component via a data interface.

- **R4 - Context Tool-Support**

  The approach should provide a tool-support for context management aspects such as modeling, transformation and execution. The tool-support should enable the modeling of various contextual parameters and their transformation to an executable context service which enables context monitoring at runtime.

UI Adaptation:

- **R5 - Adaptation Modeling Approach**

  R5.1 - Adaptation Modeling: The approach should enable the abstract specification of UI adaptation rules complementary to an existing abstract UI model. The adaptation model should be integrated in the overall modeling environment of self-adaptive UIs offering a specific viewpoint for modeling adaptation aspects. The adaptation modeling language should be also extensible and ease the developers work in specifying and maintaining UI adaptation rules.

  R5.2 - Adaptation Aspects: The approach should support various UI adaptation aspects such as content-, navigation-, presentation, and modality adaptation. The adaptation modeling approach should support various UI adaptation techniques such as content-, navigation-, presentation, and modality adaptation.

- **R6 - 'Adaptation' Transformation Approach**

  The approach should come up with a generative approach where an automated transformation from adaptation models to executable *Adaptation Services* is supported. Thus, the amount of repetitive manual code for implementing *Adaptation Services* should be reduced.

- **R7 - Runtime Adaptation**

  The approach should support runtime UI adaptation where the underlying UI is automatically changed as a reaction to context changes.

- **R8 - Adaptation Tool-Support**

  The approach should provide a tool-support for modeling, transformation or execution of self-adaptive UIs. The tool-support should enable the modeling of various adaptation techniques and their transformation to an executable *Adaptation Service* which enables UI adaptation at runtime.

Usability Evaluation:

- **R9 - End-user Satisfaction Analysis**

  The approach should consider usability evaluation regarding end-user satisfaction. The usability evaluation method should enable the assessment and analysis of end-user satisfaction.

- **R10 - Instant User Feedback**

  The approach should incorporate instant user feedback for usability evaluation. The usability evaluation method should consider instant user feedback to loop-in the end-users in the evaluation process. By collecting explicit feedback from the end-users more insights about user acceptance can be gathered.

- **R11 - Usability Test at Runtime**

  The approach should address usability evaluation at runtime while the user interacts with the system. The usability evaluation method should work while the users are interacting with the interactive system and the UI adaptations are happening at runtime.

The previously described set of requirements represents the primary focus of this thesis to enable a basic setup for development and evaluation of self-adaptive UIs. Beside that, further forms of quality assurance regarding the development and evaluation are possible which are discussed later on in Section 8.3.

3.3 Related Work

In this section, we present and discuss the state-of-the-art approaches concerning context management, UI adaptation and usability evaluation related to self-adaptive UIs.

3.3.1 Context Management

Various approaches in the area of context-aware computing were presented in the past years to deal with the topic of context management. An important architecture for building context-aware applications was already presented by Dey et al. [DAS01]. They developed a context toolkit that enables rapid prototyping of context-aware applications. The architecture of their context toolkit consists of sensors to collect context information, widgets to encapsulate the contextual information and provide methods to access the information, as well as interpreters to transform the context information into high-level formats that are easier to handle. Beside this kind of works which focus on architectural aspects of context management, various other frameworks and concrete approaches exist. For evaluating the strength and shortcomings of related approaches concerning context management, we use the previously introduced criteria R1-R4. The tabular overview in Figure 3.4 shows the main results regarding the evaluation of context management approaches.

The *Context-Aware Application Development Approach (CAADA)* [JDB18], as the name already implies, is a model-driven approach for context-aware applications. It supports the generation of code for context monitoring classes in Java. The CAADA architecture consists of three main components, *Context Management*, *Context Change Management*, and *Adaptation Management*. The framework which builds upon this architecture is called Dynamic Observation and Notification (DONCIR). It enables the creation of an extensible context model based on a context metamodel, which also includes adaptation rules. This context model is transformed to XML and then to Java classes, which can be integrated in other applications. The framework facilitates the integration of manual code to access context information. DONCIR also provides an Eclipse plug-in for creating context models.

Legend

●	Completely fulfills
◖	Partially fulfills
○	Does not fulfill

	Criteria						
	R1.1: Context Modeling	R1.2: Context Aspects			R2: Trans-Formation Approach	R3: Runtime Monitoring	R4: Tool-Support
		User	Platform	Environment			
CAADA [JDB18]	●	◖	◖	◖	●	●	●
CA-PSCF [AYG10]	●	◖	◖	◖	◖	●	●
JCAF [Bar05]	◖	○	○	○	○	●	◖
MAIS-WebML [CDMF07]	●	●	●	○	○	●	●
TriPlet [MV13]	●	●	●	●	○	○	○
WildCAT [DL05]	◖	○	○	○	○	●	◖

Fig. 3.4 Evaluation of context management approaches

The CAADA approach is quite similar to our approach as it supports the modeling, transformation and execution phases for context management. However, there are some differences to our approach. Although CAADA provides a metamodel for context modeling, the granularity of the context modeling elements are quite abstract and not directly suitable for specifying contextual parameters for UI adaptation purposes. Hence, the relevant context aspects user, platform, and environment are not explicitly covered, so that usage of this context model could hamper the straightforward specification of relevant contextual parameters.

The *Context-Aware Pervasive Service Creation Framework* (CA-PSCF) [AYG10] is an approach that allows the generation of Java based applications templates. It consists of a *Context Modeling Framework* (CMF). This is realized based on the existing *Eclipse Modeling Framework* (EMF), which provides generator components. The CMF embraces a context modeling language for creating context models. The model can then be transformed to *Context Services*, but their functionality still has to be implemented. CA-PSCF provides an Eclipse plugin for supporting context modeling.

Similar to the previously described approach, CA-PSCF supports the modeling of contextual parameters through a dedicated modeling language. However, this approach also does not support explicit modeling of context aspects such as user, platform, and environment. Furthermore, it is important to mention that the transformation approach is only supporting the generation of stubs where interfaces for the monitoring classes are created. Therefore, additional manual code is needed for establishing the context monitoring functionality.

JCAF [Bar05] and *WildCAT* [DL05] were introduced to support the development of context-aware applications. Both, WildCAT and JCAF are frameworks based on the programming language Java and they support context management by allowing the definition of a dynamic data model to represent the execution context for several application domains. In addition, they offer a programming interface to discover, interpret and monitor the events occurring in an execution context and record every change occurring in the context model. Both, JCAF and WildCAT facilitate the creation of a context model through a data model. As the approaches are programming-based, the context model is directly represented as source code and does not enable the abstract specification of contextual parameters. Also relevant context aspects like user, platform, and environment are not explicitly covered in these approaches. A transformation approach for generating *Context Services* is not needed as the context definition and handling is represented as source code. Main drawback of both approaches is that they are primarily developed for Java applications and do not provide fine grained context monitoring features for supporting UI adaptation.

Another approach for context management is MAIS-WebML [CDMF07], pursued in the project MAIS in the context of the web modeling language. It embraces a conceptual framework that provides modeling facilities for context-aware web applications. It consists of a data scheme and a definition of a website structure and behavior. The approach expands the IFML predecessor WebML for modeling context information and the web application itself. The approach provides tooling for context modeling based on an editor called CASE. MAIS-WebML enables the modeling of context, but does not support fine grained specification of context aspects such as user or platform. The context aspect environment is out of scope in this approach. The generation of code for *Context Services* is also not considered in MAIS-WebML.

While the before mentioned approaches consider context management for a broad spectrum of applications areas, there are also specific approaches that deal with context management and context modeling specifically for supporting the adaptation of user interfaces of interactive systems. One holistic approach in this direction is the conceptual framework named TriPlet [MV13]. It consits of a *Context-Aware Meta-Model* (CAM), a *Context-Aware Reference Framework* (CARF) and the *Context-Aware Design Space* (CADS). The CAM is the basis for modeling contexts in this approach. It includes the triplet user, platform, environment and allows context modelers to extend more properties. Other components are the *Context-Aware*

Reference Framework (CARF), a framework for listing relevant concepts as a guide for implementation, and the *Context-Aware Design Space* (CADS), for analyzing, comparing and evaluating context-aware applications. As TriPlet is limited to a conceptual framework for context-aware applications, it does not support the model-driven generation of code for *Context Services* and it is also does not provide any tool-support to enable runtime monitoring of context information.

In summary, existing approaches related to context management does not fully support a model-driven context management solution for self-adaptive UIs which enables the specification, automatic generation of *Context Services*, and context monitoring at runtime.

3.3.2 UI Adaptation

In recent research, adaptive or self-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 [ABY14a]. A key goal behind self-adaptive UIs is plasticity denoting a UI's ability to preserve its usability despite dynamically changing context-of-use parameters [Cou10]. In practice, especially in the context of web design, the paradigm of Responsive Web Design (RWB) is widely used to adapt the layout of a web page in response to the characteristics of the used device. Furthermore, there are several approaches, which adopt and apply the idea of adaptive UIs for different domains like health care systems [SRS15], enterprise applications [Aki14] or smart working environments [GMP$^+$15].

As a survey [ABY14a] on adaptive model driven UI development systems shows, model-based and model-driven engineering formed the basis for most of the systems targeting the development of self-adaptive UIs. In the following, we present an overview of existing frameworks and concrete approaches dealing with the topic of UI adaptation. For evaluating the strength and shortcomings of those state-of-the art approaches and to compare them against our UI adaptation approach, we used the previously introduced criteria R5-R8. The tabular overview in Figure 3.5 shows the main results regarding the evaluation of UI adaptation frameworks and approaches.

A *3-Layer Architecture* was presented by Lehmann et al. [LRBA10] for devising adaptive smart environment user interfaces. The layered approach represents the life cycle of context-adaptive applications and the scope of context information at runtime. The authors of this work, provide a model-based implementation of the proposed architecture which is based on the idea of executable models. Adaptation modeling is not covered with a dedicated modeling

Legend
● Completely fulfills
◐ Partially fulfills
○ Does not fulfill

		Criteria							
		R5.1: Adaptation Modeling	R5.2: Adaptation Aspects				R6: Trans-formation Approach	R7: Runtime Adaptation	R8: Tool-Support
			Content	Navigation	Presentation	Modality			
Frameworks	3-Layer Architecture [LRBA10]	◐	○	○	●	●	○	●	◐
	CAMELEON-RT [BDB+04]	○	○	○	○	○	○	◐	◐
	CEDAR [ABY12]	◐	●	○	●	○	○	●	●
	FAME [DC06]	◐	◐	○	◐	●	○	◐	◐
	Malai [BB10]	◐	●	●	●	○	◐	◐	●
	TriPlet [Mot13]	◐	●	●	●	●	○	◐	○
Approaches	AOM [BMB+11]	◐	●	○	●	○	◐	●	●
	Comet(s) [CCD+04]	◐	●	○	●	○	○	●	○
	DynaMo-AID [CLV+03]	◐	●	○	●	○	◐	●	◐
	MASP [FBA06]	◐	○	○	●	●	◐	●	●
	Mining Minds [HHB+18]	◐	●	●	●	●	○	●	●
	MyUI [PHJS12]	◐	◐	●	●	●	◐	●	●
	RBUIS [ABY13]	◐	●	○	●	○	○	●	●
	Supple [GWW10]	○	○	○	●	○	○	●	●

Fig. 3.5 Evaluation of UI adaptation frameworks and approaches

language which complements an existing abstract UI modeling language like IFML. Also the focus is more on the adaptation aspects of making layout changes addressing presentation and modality. The *3-Layer Architecture* does not focus on the generation of executable code for the adaptation logic, instead executable models are used for realizing the adaptive behavior. Although the approach was implemented, there is no specific tool-support for devising and maintaining self-adaptive UIs in general.

CAMELEON-RT [BDB$^+$04] is a reference architecture model for distributed, migratable, and plastic user interfaces within interactive spaces. It follows the CRF and supports all abstraction layers. CAMELEON-RT serves as a high-level reference but does not provide a concrete language for adaptation modeling nor a transformation approach for generating *Adaptation Services*. The runtime UI adaptation aspects are explained more on a conceptual level so that a concrete instantiation is missing. As a proof-of-concept implementation, the authors present a middleware solution for realizing UI adaptation. However, concrete tool-support for modeling, transformation and execution of self-adaptive UIs is not provided.

CEDAR [ABY12] is a reference architecture for stakeholders interested in developing adaptive enterprise application UIs based on an interpreted runtime model-driven approach. This means CEDAR supports UI adaptation based on the interpretation of UI models at runtime. Hence, generation of code for *Final UI*s and *Adaptation Services* is not necessary. The goal of CEDAR is to utilize the dynamic nature of the models to improve the flexibility of the UI adaptation. In general, CEDAR supports UI adaptation modeling, however the involved UI modeling languages are not based on de facto standards such as IFML. The reusability of the adaptation modeling approach proposed in CEDAR could be problematic in other environments as it is highly adopted to the own specific languages and tool-kit. CEDAR is considering UI adaptation aspects like content and presentation, however there is no explicit support for navigation- and modality-adaptation.

FAME [DC06] is a model-based Framework for Adaptive Multimodal Environments. It proposes an architecture for adaptive multimodal applications and provides means to represent adaptation rules - the behavioral matrix - and a set of guidelines to assist the design process of adaptive multimodal applications. FAME's practical benefit was shown based on the development process of an adaptive Digital Talking Book player. The primary focus of FAME is modality adaptation hence it is not meant to be a general-purpose reference framework for adapting other UI characteristics. Therefore, adaptation aspects such as navigation or presentation are not or only partially covered. Similar to the above described frameworks FAME does not provide a dedicated UI adaptation modeling that is conform to UI modeling standards.

Malai [BB10] is an architectural model for interactive systems and forms a basis for a technique that uses aspect-oriented modeling (AOM) for adapting user interfaces. Beside finite state machine diagrams, a domain specific language called the Malai language is utilized to specify different UI and adaptation aspects. Supported UI adaptation aspects are content, navigation, and presentation while modality is not covered. Malai enables the generation of UI code (e.g., Swing, .NET), however it does not primarily focus on generating code for *Adaptation Services*.

TriPlet [Mot13] is a computational framework for context-aware adaptation and consists of a metamodel, a reference framework and adaptation aspects for adaptive UIs. Based on the extensive systematic review of existing work, the authors propose a context-aware adaptation

(CAA) framework that covers different aspects such as content, navigation, presentation and modality. Although TriPlet introduces a metamodel for CAA, it does not provide a dedicated modeling langauge for UI adaptation. Also, a generative approach which is enabling the transformation of self-adaptive UIs is not covered. There are some proof-of-concept case studies showing the benefit of CAA and adaptive UIs. However, a generic and practically usable tool-support for devising self-adaptive UIs is missing.

AOM [BMB+11] is a concrete UI adaptation approach which is build up on the basis of the before described Malai architecture. It combines aspect-oriented modeling with property-based reasoning. The encapsulation of variable parts of interactive systems into aspects permits the dynamic adaptation of user interfaces. Tagging UI components and context models with QoS properties allows the reasoner to select the aspects the best suited to the current context. As this approach is based on the idea of aspect-oriented modeling it does not introduce a holistic UI adaptation modeling. Instead, content and presentation related UI adaptation aspects are weaved into an existing application to support these UI adaptations at runtime. Due to the same reason, AOM does not focus on a transformation approach where code for UI *Adaptation Services* is generated.

The COntext Mouldable widgET (Comet(s)) [CCD+04] approach was introduced as a set of widgets that support UI plasticity. It provides an architectural-style for plastic UIs by combining the toolkit and model-based approaches presented in [DCC08]. A 'Comet' is an introspective widget that is able to self-adapt to some context-of-use, or that can be adapted by a tier-component to the context-of-use, or that can be dynamically discarded (versus recruited) when it is unable (versus able) to cover the current context-of-use. The novelty in this approach is to support UI adaptation at widget level and to decentralize the adaptation logic. Although this approach enables high reusability of various adaptive widget components which support the efficient development of self-adaptive UIs, it does not introduce a UI adaptation modeling language for describing the adaptation logic behind the adaptive widgets. Similar to the AOM approach it focuses on the adaptation aspects content and presentation while navigation and modality are not explicitly supported. A transformation approach is not considered in the Comet(s) approach as it is more relying on the paradigm of component-based development.

DYNAmic MOdel-bAsed user Interface Development (DynaMo-AID) [CLV+03] is a design process and runtime architecture for devising context-aware UIs and is part of the Dygimes UI framework [CLV+03]. Its runtime architecture includes three major modules namely context monitoring, functional core, and presentation that are linked by a dialog controller. The final UI is rendered from task models after adapting them to the operating environment and device. The DynaMo-AID approach does not provide a dedicated UI adaptation modeling language. Instead, adaptations are described on the *Tasks and Concepts* layer and out of the task model the generation of final UIs is supported. The approach mainly focuses on device specific UI adaptations while context-of-use information about user and interaction environment are not explicitly covered. The provided tool-support is tailored to the specific application scenario and does not support the development of self-adaptive UIs in general.

The Multi-Access Service Platform (MASP) [FBA06] is a UI management system targeting ubiquitous UIs for smart homes. MASP uses a model-based approach to develop and adapt UIs based on task-tree models similar to the previously mentioned approach. Hence, it does not provide a dedicated UI adaptation modeling language as a complementary language for standardized UI modeling languages such as IFML. The main focus of supported UI adaptation aspects are presentation and modality. The used transformation approach is specific in the sense that it is focusing on the transformation of task-tree models to final UI code.

The Mining Minds platform [HHB+18] proposes a model-based development methodology for adaptive UIs. The proposed methodology is implemented as an adaptive UI authoring tool which is called A-UI/UX-A. A-UI/UX-A is a system capable of adapting user interfaces based on the utilization of contextual factors, such as user disabilities, environmental factors (e.g., light level, noise level, and location) and used device. The Mining Minds approach introduces models for specifying user, context and device characteristics. However, there is not an explicit UI adaptation modeling workbench supported in the A-UI/UX-A tool-chain. Instead, UI adaptation rules are specified in an event-condition-action-manner based on a textual description. Compared to our approach the Mining Minds approach focuses on a model-based development process rather than a model-driven development approach. Hence, the criteria transformation approach is not fulfilled.

MyUI [PHJS12] is a user interface development infrastructure for improving accessibility through adaptive UIs. It uses an open pattern repository for defining adaptation rules. User interfaces are specified as an abstract model that is represented using a notation based on state charts. MyUI allows adaptations according to the *User Profile* and *Device Profile*. Additionally, the developer can specify customization settings in the *Customization Profile*. The User, Device and Customization profiles are transformed into the *User Interface Profile*, which defines the general characteristics of the UI. It is created at the beginning of each interaction session with an application generated with the MyUI approach. The UI itself, without any layout specifications, is defined in the Abstract Application Interaction Model (AAIM). The AAIM and the User Interface Profile are used to select the *User Interface Elements*, which suit the current context the best. While MyUI supports context and adaptation modeling in a detailed manner, it does not consider environmental context information as a basis for UI adaptation. Therefore, the expressiveness of the adaptation modeling language is limited and the adaptation aspects can not be fully covered. In [PHJS12], the authors write that they support the generation process of adaptive UIs based on the presented MyUI concept, however transformation specific details are not provided in detail.

Role-Based UI Simplification (RBUIS) [ABY13] is a mechanism for improving the usability of enterprise application UIs by providing users with a minimal feature-set and an optimal layout based on the context-of-use. As RBUIS is based on the CEDAR architecture the same argumentation for the evaluation criteria is valid for RBUIS. It has to be noted that CEDAR and RBUIS with its tool-support called CEDAR studio form a complementary solution where the benefit of adaptive UIs have been shown. For this purpose, the authors used their approach to integrate UI adaptation features into an open source ERP legacy system called OFBiz. From this perspective, this approach is one of the pioneering approaches for devising adaptive UIs in practical usage scenarios. However, main differences of this approach compared to ours is that they use an interpretative approach while ours is generative. Also this approach is not focusing extensively on a separate way of modeling UI adaptations and context as it is in our case. Hence, our approach also supports context-management in a more detailed and easy-to use manner as we can can generate *Context Services* to monitor different context-of-use situations while this is not possible in RBUIS.

Supple [GWW10] supports automatic generation of UIs adapted to each user's abilities (e.g., motor and vision), devices, tasks, and preferences. It relies on a high-level interface specification, device model, and user traces to generate the UI. The only adaptation type supported by Supple is layout optimization. Vision and motor capabilities are the primary supported adaptation aspects, and 40 UI factors (e.g., font size, widget size, etc.) are supported. Supple does not provide a means for extending adaptation types, aspects, and factors. Also, it has been criticized for exceeding acceptable performance times. This criticism could be justified by observing some of its worst-case scenarios that could span over 30 seconds when computing the most appropriate UI layout. This timing is not appropriate for software systems looking for high efficiency. One advantage that Supple has over other systems lies in performing true layout optimization due to its ability to quantify UI quality. The quantification is achieved by using a cost function to compare UI versions in order to determine the most optimal one. This approach also allows Supple to support trade-off analysis, which was demonstrated for a fixed number of adaptation aspects, namely motor and vision capabilities. Supple supports no explicit means for UI adaptation modeling based on a domain specific language, instead it models users in terms of actual traces. As described above mainly layout optimizations are considered, hence the main focused adaptation aspect is presentation. As in the context of Supple, the authors consider interface generation as an optimization problem, there is not a classical model-driven UI development approach behind where code is generated.

Beyond the above discussed frameworks and approaches which can be categorized as rule- and optimization- based approaches, there are also learning-based approaches like [HTLK08] which uses machine learning or [BMB$^+$11] where a genetic algorithm is used to calculate a well suited UI adaptation.

In summary, existing approaches related to UI adaptation does not fully support a model-driven engineering solution for self-adaptive UIs where context management and UI adaptation concerns are supported in an integrated manner. Also, most of the existing approaches are focusing on specific UI adaptation aspects without integrating them together. Finally, it has to be noticed that most of the existing UI adaptation approaches are model-based and do not explicitly focus on automated generation of UI *Adaptation Services*.

3.3.3 Usability Evaluation

As already described in prior work [FG09], usability evaluation of adaptive UIs is a challenging task that should be addressed by adequate methods and techniques. A literature review [vVvdGKS08] is showing that mostly *questionnaires*, *interviews*, and *data log analysis* methods are used for evaluating adaptive UIs. However, the authors argue that these methods are not fully suitable to investigate the usability of adaptive UIs due to quality of questionnaires, shallow state of think aloud protocols, or the need for triangulation of the data logs to derive useful information.

With regard to existing usability evaluation approaches, we can observe that various usability evaluation criteria such as ease of use, learnability, task performance time, accuracy, predictability, or satisfaction were investigated. In [PLN04], for example, the authors investigate usability trade-offs for adaptive UIs with regard to ease of use and learnability. To this end, a user support concept was developed and applied to a context-aware mobile device with an adaptive UI. The approach was evaluated based on a usability test and main results show that the user support improved ease of use, but unexpectedly it reduced learnability of the adaptive UI. Moreover, in [LM10] for example, the authors examine the positive and possible adverse effects of adaptive UIs in the context of an in-vehicle telematic system. For this purpose, they conduct a usability experiment with different participants to measure the task performance time of the users using the proposed adaptive UI. Other studies such as [GET$^+$08] are examining the relative effects of predictability and accuracy on the usability of adaptive UIs. The results of this study show, for example, that increasing predictability and accuracy lead to strongly improved satisfaction.

Beside above mentioned classical usability evaluation methods, prior work in the field of plastic user interfaces also investigated the question how the notion of mappings as promoted by *Model Driven Engineering (MDE)* can be exploited to control UI adaptation according to explicit usability criteria. In [SCCF07], for example, the authors present a formal mapping between source and target models specifying the usability properties that are preserved when transforming source models into target models. However, the applicability of such formal methods in real application scenarios is quite difficult as ergonomic criteria for ensuring usability may be inconsistent, and as a result, require difficult trade-offs [SCCF07].

More recent approaches from the area of mixed-initiative interfaces [BCM07], such as *Crow-dAdapt* [NSN13], make use of a combination of end-user development and crowd-sourcing for context-aware adaptation of UIs. Furthermore, [Mez13] suggests to incorporate user feedback via a promoting/demoting technique to further improve UI adaptations. While this approach mainly focuses on UI adaptation improvement based on a combination of user feedback and machine learning, the idea of collecting instant user-feedback can be also used to test the end-user satisfaction of UI adaptations. As our research question (see Subsection 1.2) is indicating, an important research topic for our work is to investigate the acceptance of self-adaptive UIs by collecting feedback from the users. In this sense, our work differs from current research done in the field. For a systematic comparison of our solution with the existing related work, we used the previously introduced criteria R9-R11. The tabular overview in Figure 3.6 shows the main results of the comparison.

AppEcho [SOB14] is a user-driven, in situ feedback approach for mobile platforms and applications. The authors present a mobile feedback approach, which enables users to document individual feedback on mobile systems in situ. The collected information can then be evaluated and used as new requirements by developers. While this approach explicitly targets mobile platforms and uses instant user feedback mechanisms as in our solution, it is only

Legend

● Completely fulfills
◖ Partially fulfills
○ Does not fulfill

	Criteria		
	R9: End-user Satisfaction Analysis	R10: Instant User Feedback	R11: Usability Test at Runtime
AppEcho [SOB14]	◖	●	○
AUE [FBK+08]	○	◖	●
AUI Mobile [MM02]	○	○	○
Media Maps [WSvT10]	●	○	○
MOCCA [RB11]	●	○	○
Mining Minds [HHB+18]	●	●	○
OSApp [AMI17]	●	○	○
RBUIS [ABY13]	●	◖	○

Fig. 3.6 Evaluation of usability evaluation approaches for self-adaptive UIs

partially focusing on end-user satisfaction analysis. With this regard, the *AppEcho* approach is providing a more generic feedback mechanism which is not explicitly focusing on the evaluation of UI adaptations and their effect on user satisfaction based on the context-of-use. As the *AppEcho* approach aims to provide a generic feedback mechanism to collect a large amount of individual user feedback about mobile applications, it is not designed with the purpose to support usability testing for self-adaptive UIs at runtime.

Another related approach, called *Automated Usability Evaluation* (AUE) of model-based interactive systems, is presented in [FBK+08]. In this paper, the authors describe an approach to efficiently evaluate the usability of an interactive application that has been realized to support various platforms and modalities. By using a model-based runtime environment incorporating a mental model of the end-user, the authors aim to reduce the evaluation effort by automating parts of the testing process for various combinations of platforms and user groups. While this approach supports usability testing at runtime and also targets mobile platforms, it is only partially covering the aspect of instant user feedback. As the user is not looped in to the real interaction scenario and is represented through a model, an explicit feedback of the real end-users cannot be collected. In contrast to our solution, this approach is also not focusing on the usability aspect end-user satisfaction.

AUI Mobile [MM02] is a further approach which is investigating the application of adaptive user interfaces for mobile platforms. In this work, the authors propose a solution for the adaptation of graphical user interfaces by using a mobile agent system. Applicability of this approach is shown based on a mobile currency converter and a survey application. This approach mainly focuses on the application of adaptive UIs for mobile platforms. However, this work is not focusing on usability evaluation, instant user feedback or usability testing of UI adaptation features at runtime.

A usability evaluation study of adaptive UIs for mobile platforms is presented in [WSvT10]. In this work, the main research question is whether adaptive UIs improve the usability of mobile applications. For this purpose, the authors discuss several simple types of UI adaptations based on a case study dealing with *Media Maps*. Based on this case study, they conduct a usability test with 20 participants with a post-test satisfaction questionnaire. The main results of this usability study show that adaptive UIs have potential to improve mobile applications regarding accuracy and end-user satisfaction. Compared to our solution, the described usability study does not incorporate instant user feedback nor establishes a way to efficiently test the end-user satisfaction rate of individual users based on their current context-of-use.

A deeper analysis of various usability aspects such as performance, perceived usability, and aesthetics is presented in the *MOCCA* [RB11] approach, which is focusing on culturally adaptive user interfaces. In this work, the authors argue that it is not feasible to design one interface that appeals to all users of an increasingly global audience. Instead, they propose to design culturally adaptive systems, which automatically generate personalized interfaces that correspond to cultural preferences. Based on a usability test with 41 participants of different cultural backgrounds, they demonstrate the benefit of the approach by showing that the majority preferred their culturally personalized interfaces to a non-adapted version. Moreover, this study shows that the participants were able to work 22% faster with the culturally adapted interface, showing that the approach improves both the performance as well as the user satisfaction. While the *MOCCA* approach analyzes the usability of culturally adaptive UIs for different platforms such as mobile or desktop and also focuses on the aspect of user satisfaction, it is not incorporating instant user feedback for efficiently testing the usability of different UI adaptations at runtime. However, this is essential for efficiently conducting long-term usability studies about adaptive UIs with a large amount of participants.

The *Mining Minds* [HHB$^+$18] platform is a further approach which supports UI adaptation and provides a detailed usability evaluation based on context and user experience. The authors of this work present a domain and platform independent model-based methodology for devising adaptive user interfaces. The methodology is implemented as a tool capable of adapting the user interface based on contextual factors. The *Mining Minds* platform supports various UI adaptation techniques like task-feature set, layout, and modality adaptation for different platforms including mobile devices. In addition, an instant user feedback mechanism is integrated in this approach. However, the collected user feedback is more general and does not allow to rate specific UI adaptations in a specific context-of-use. Furthermore, the authors do not provide evaluation results based on the collected user feedback data, but rather present the results of a usability experiment with 32 participants based on a questionnaire. Although the *Mining Minds* platform is the most similar approach to ours, the main difference is that our approach supports a fine grained rating for triggered UI adaptation features reflecting the current context-of-use at runtime. Also in our solution, we use the collected context and corresponding user feedback data to perform a data-driven satisfaction analysis of UI adaptation features.

In [AMI17], the authors present a context-aware adaptation approach for mobile applications driven by software quality and user satisfaction. The proposed approach uses a reward formula, containing the notion of software availability and the user's feedback to determine the best adaptation that needs to be applied in a given context. The feasibility of this approach is shown based on a mobile application, called Off Site Art *OSApp*. As a result of their approach, the authors provide a reward formula by describing the user satisfaction model and user-perceived service availability model. While this approach covers end-user satisfaction analysis and targets mobile platforms, it does not focus on instant user feedback and usability testing at runtime. However, the introduced reward formula in this work could be used to also assess the usability of adaptive user interfaces for mobile platforms.

Role-Based UI Simplification (RBUIS) [ABY13] is a further approach for supporting adaptive UIs. The authors of this work define UI simplification as a mechanism for increasing usability through adaptive behaviour by providing users with a minimal feature-set and an optimal layout based on the context-of-use. The *RBUIS* approach mainly focuses on simplification of enterprise applications accessed via a desktop computer, but also considers mobile device usage. The authors evaluated the approach using an online interactive survey with a UI pair composed of an initial and a simplified UI. The main results of the user study with 25 participants show that simplifying enterprise application UIs based on roles improves user satisfaction and efficiency. Although user feedback is partially integrated in the *RBUIS* approach by allowing users to *Apply and Keep* and *Apply Once* of UI simplification operations, it is not used for explicitly testing the acceptance of specific UI adaptation features at runtime. In contrast to our solution, *RBUIS* is also not directly mapping the current context-of-use to the provided feedback by the users.

In summary, the existing approaches and usability evaluation methods are not fully addressing the analysis of end-user satisfaction for self-adaptive UIs by combining instant user feedback and usability testing of UI adaptation features at runtime.

3.4 Summary

As the evaluation of related work shows, there are still limitations in the state-of-the-art approaches for supporting the development and evaluation of self-adaptive UIs.

Concerning the context management aspect, we can sum up that various context modeling approaches already exist. However, most of the practically suitable approaches which provide tool-support are not sufficient to enable context modeling for self-adaptive UIs in a fine-grained manner. The existing approaches which focus directly on context management for UI aspects are more on conceptual level or do not provide sufficient tool-support to ease the work of developers in specifying different contextual parameters as an essential prerequisite for UI adaptation.

The evaluation of existing UI adaptation frameworks and approaches shows that the state-of-the-art still has some improvement potential to consolidate the various UI adaptation aspects in an integrated adaptation modeling language. Although various approaches already provide tool-support for UI adaptation, an integrated model-driven development approach supporting UI, context, and adaptation concerns is missing.

A similar picture can be seen when we look in to the evaluation results of the usability evaluation approaches for self-adaptive UIs. A comprehensive and flexible solution for runtime usability evaluation of self-adaptive UIs regarding end-user satisfaction is missing. Especially a usability evaluation solution approach for self-adaptive UIs which is incorporating the current context-of-use enriched with instant user feedback has not been addressed in previous works.

All in all, based on the detailed analysis of related work, we can conclude that a model-driven engineering approach is required to support the development and usability evaluation of self-adaptive UIs as well as to address the drawbacks of the existing approaches.

Chapter 4

Modeling

This chapter describes our general approach to the modeling of self-adaptive user interfaces. In Section 4.1, we describe the overall language engineering approach that we have taken. In Section 4.2, we give an overview of our integrated modeling framework for self-adaptive UIs. In Section 4.3, we introduce our context modeling language *ContextML*, which supports the specification of various heterogeneous context-of-use situations. After that, in Section 4.4, we introduce our adaptation modeling language, called *AdaptML*, which supports the specification of different UI adaptation rules that can be triggered at runtime in order to automatically adapt the UI to the changing context-of-use parameters. Section 4.5 concludes this chapter with a summary and discussion.

4.1 Language Engineering Approach

Modeling self-adaptive UIs is a challenging task as cross-cutting concerns such as context management and UI adaptation have to be treated appropriately and separately from the business logic. To address this issue and reduce the complexity in development of self-adaptive UIs, a novel integrated modeling framework is needed which covers UI, context, and adaptation modeling aspects in a uniform manner.

The integrated modeling framework that is presented in this chapter was systematically derived from several information sources. The approach used for language engineering is depicted in Figure 4.1.

On the top of this figure, the *High-level Language Requirements* are depicted. According to our modeling framework for self-adaptive UIs, we have identified the following *High-level Language Requirements*:

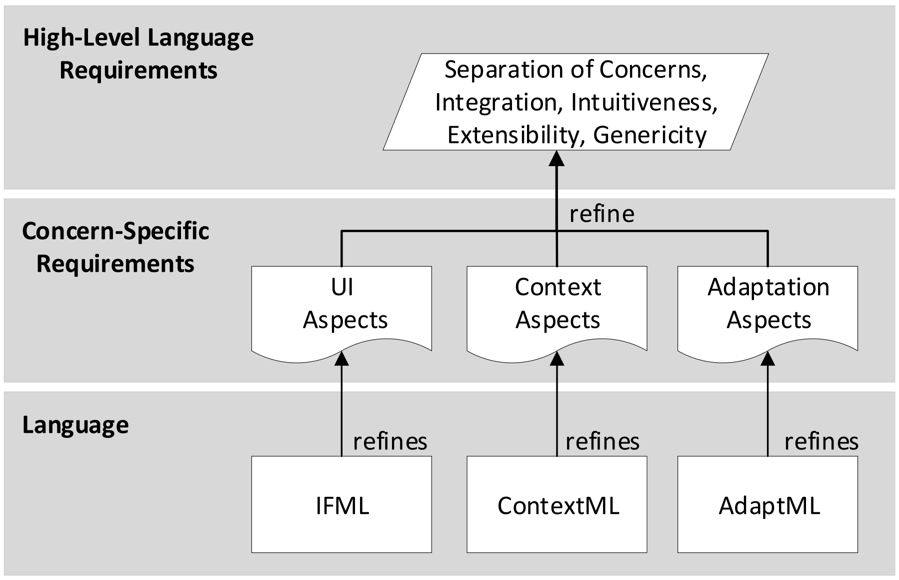

High-Level Language Requirements
Separation of Concerns, Integration, Intuitiveness, Extensibility, Genericity
refine
Concern-Specific Requirements
UI Aspects
Context Aspects
Adaptation Aspects
Language
refines
refines
refines
IFML
ContextML
AdaptML

Fig. 4.1 Language Engineering Approach

- *Separation of Concerns*: To allow focusing on the relevant concerns such as UI, context, and adaptation within a software system. These concerns should be separated from the core business logic concerns. That is, a modeling approach should provide a dedicated modeling view to specify UI, context, and adaptation aspects as well as abstracting from the rest of the system's logic.
- *Integration*: To best leverage a method that supports the modeling of self-adaptive UIs, this method should be integrated into existing software engineering methods. In the domain of model-driven UI development, that is, a developer can use the modeling approach in combination and conform to existing standard (UI) modeling languages from OMG.
- *Intuitiveness*: To increase acceptance, the approach should be intuitive to use, especially since early in the engineering process, non-experts might be required to model and/or understand the context-of-use and UI adaptation specification. To support the intuitiveness, well-known techniques and paradigms should be reused.
- *Extensibility*: To increase the flexibility, the modeling approach should be extensible so that new aspects related to context-management or UI adaptation can be easily added when it is required.

- *Genericity*: As known from general purpose languages such as the UML, the approach shall be applicable in a variety of domains. The generic approach should support the specification of self-adaptive UIs on an abstract level using different degrees of detail.

Based on the analysis of the problem domain (see Section 3.2), the *High-level Language Requirements* were refined to *Concern-Specific Requirements*. The *Concern-Specific Requirements* can be divided into three parts: *UI Aspects*, *Context Aspects*, and *Adaptation Aspects*. Regarding *UI Aspects*, the concrete concern-specific requirement was basically to model a user interface by characterizing its content, structure, and navigation in an abstract manner. Regarding *Context Aspects*, the main concern-specific requirement was to support the holistic specification of various context-of-use situations covering especially the context parameters user, platform, and environment. Finally, the concern-specific requirement for *Adaptation Aspects* was to enable the specification of UI adaptations through describing conditional expressions and characterizing various UI adaptation techniques such as task-feature set-, layout-, or navigation adaptation.

To fulfill the above described requirements, we decided to structure our integrated modeling framework for self-adaptive UIs into three main modeling view points: UI, context, and adaptation (see *Language* layer at the bottom of Figure 4.1). Each modeling view point is supported through a dedicated domain-specific language (DSL) to be conform to the paradigm of separation of concerns. To comply with the second high-level language requirement, we decided to use the *Interaction Flow Modeling Language (IFML)* as an OMG standardized UI modeling language. Based on *IFML*, we developed two complementary modeling languages: *ContextML* for specifying various heterogeneous context-of-use situations and *AdaptML* for specifying UI adaptations. In the design process of *ContextML* and *AdaptML*, we took special care for the high-level language requirements *Intuitiveness*, *Extensibility*, and *Genericity* and also for the mentioned concern-specific requirements which will be explained in detail when the according DSLs are introduced.

A further important design decision for our language engineering approach was to focus on the abstract UI layer. Although the CAMELEON reference framework (see Section 2.1.3) ideally foresees four different abstraction layers in the model-driven UI development process, our approach is not explicitly covering the *Tasks and Concepts* layer as there is no flexible solution for automatically deriving *IFML* abstract UI models based on common task modeling languages such as CTT [PMM97] or Hamsters [PM15]. Also, an explicit modeling layer for the *Concrete UI (CUI)* is not required in our model-driven engineering approach as it is primarily focusing on web-based applications (see Section 5 for more details). Finally, it should be noticed that our language engineering approach focuses on the basic aspects UI, context, and adaptation for modeling self-adaptive UIs, while other relevant aspects such as business logic or data management are out of scope.

4.2 Modeling Framework for Self-adaptive UIs

This section gives an overview of our integrated modeling framework and provides an example for modeling self-adaptive UIs based on our running example *LibSoft* introduced in Section 3.1.

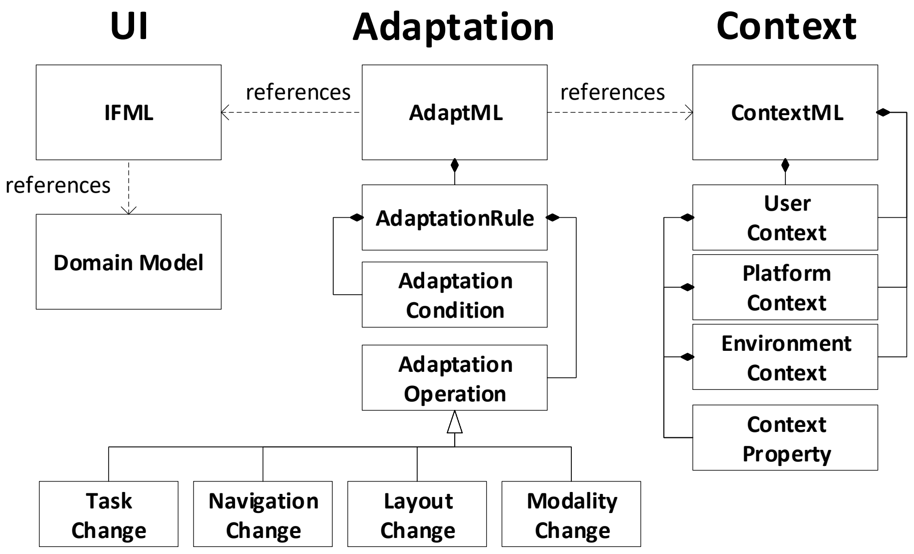


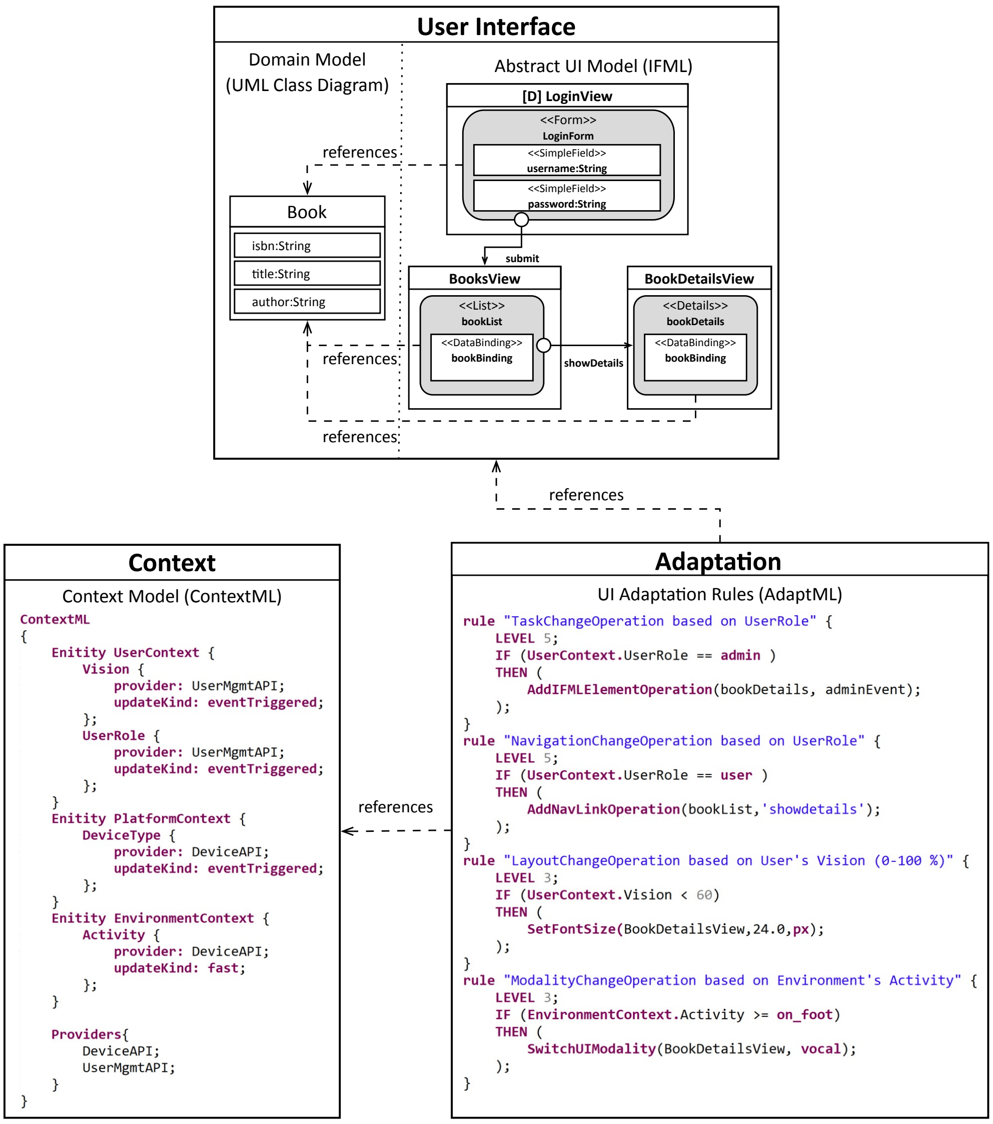
Fig. 4.2 Overview of the Modeling Framework

As shown in the overview in Figure 4.2, our modeling framework for self-adaptive UIs consists of three integrated modeling languages, namely *IFML*, *AdaptML*, and *ContextML*. While *IFML* is a standard modeling language for UI concerns and presumes that IFML *Domain Model* is represented as a UML clas diagram, special attention is required for context management and UI adaptation.

Context management brings additional complexity to cope with, as context information has to be captured through various sensors from heterogeneous sources and dynamically changing context-of-use parameters have to be detected. Therefore, we developed the context modeling language *ContextML*. It allows to specify different context-of-use parameters such as *UserContext*, *PlatformContext*, and *EnvironmentContext*. Each of these context-of-use parameters define a specific *ContextProperty* to capture a relevant context information through a context provider which is accessed through a hardware context sensor. Further details about *ContextML* and its structure follow in Section 4.3.

Beside *ContextML*, we developed another domain specific language, *AdaptML*, which supports the modeling of UI adaptations. *AdaptML* is designed as a complementary modeling language to OMG's core UI modeling language *IFML* and allows domain experts, for example web designers, to model adaptation concerns by specifying the conditions and actions for UI adaptations. Therefore, *AdaptML* contains *AdaptationRule(s)* where each *AdaptationRule* consists of an *AdaptationCondition* and *AdaptationOperation*. *AdaptationCondition* allows to specify the conditions for triggering UI adaptations through logical expressions. Therefore, *AdaptML* has a reference to *ContextML* to use the *ContextProperty* for defining the adaptation rule conditions. *AdaptationOperation* enables to specify a concrete UI adaptation by referencing the *IFML* model. This way, UI *AdaptationOperations* can be specified that relate to a specific UI model element. *AdaptML* supports the specification of various UI *Adaptation Operations* such as *Task Change*, *Navigation Change*, *Layout Change*, and *Modality Change*. Further details about the *Adaptation Operations* and *AdaptML* follow in Section 4.4.

In the following, we describe the general idea of our integrated modeling framework for self-adaptive UIs based on the running example *LibSoft* introduced in Section 3.1. Figure 4.3 shows a simplified modeling example for the *LibSoft* scenario, where UI, context, and adaptation concerns are specified based on *IFML* [Obj15], *ContextML*, and *AdaptML* in an integrated manner. On the top of this Figure, small excerpts of the domain model in the form of a UML [Obj17] class diagram and core UI models are depicted. There is an abstract UI model for a simplified library application based on *IFML* which shows the representation of three UI view containers *LoginView*, *BooksView* and *BookDetailsView* which are connected by the navigation edges *submit* and *showDetails*. To enable the specification of data bindings in *IFML*, the corresponding classes from the domain model are referenced, in our case it is the class *Book*. For specifying the different context-of-use parameters, in the bottom left corner of Figure 4.3, an illustrative context model based on *ContextML* is depicted. The shown *Context Model* contains different context *Entities* to characterize various context properties like *UserContext*, *PlatformContext* or *EnvironmentContext*. A specific context *Entity* is described in detail by its context property (e.g., mood of a user or used device type) which is defined though a context provider and update type. In the example context model, for instance, the *Vision* of the user is specified as a relevant context property which is gathered through user input. In this example, the context provider is the *UserMgmtAPI* which manages to prompt the user questions and store input data about user related information. Another example is shown based on the context property *DeviceType*. Here, the contet provider is the DeviceAPI which is responsible for identification of the used target device (e.g. smartphone, tablet or desktop). To support the specification of UI adaptation rules in addition to the *IFML*



```
ContextML
{
    Enitity UserContext {
        Vision {
            provider: UserMgmtAPI;
            updateKind: eventTriggered;
        };
        UserRole {
            provider: UserMgmtAPI;
            updateKind: eventTriggered;
        };
    }
    Enitity PlatformContext {
        DeviceType {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
    }
    Enitity EnvironmentContext {
        Activity {
            provider: DeviceAPI;
            updateKind: fast;
        };
    }

    Providers{
        DeviceAPI;
        UserMgmtAPI;
    }
}
```

```
rule "TaskChangeOperation based on UserRole" {
    LEVEL 5;
    IF (UserContext.UserRole == admin )
    THEN (
        AddIFMLElementOperation(bookDetails, adminEvent);
    );
}
rule "NavigationChangeOperation based on UserRole" {
    LEVEL 5;
    IF (UserContext.UserRole == user )
    THEN (
        AddNavLinkOperation(bookList,'showdetails');
    );
}
rule "LayoutChangeOperation based on User's Vision (0-100 %)" {
    LEVEL 3;
    IF (UserContext.Vision < 60)
    THEN (
        SetFontSize(BookDetailsView,24.0,px);
    );
}
rule "ModalityChangeOperation based on Environment's Activity" {
    LEVEL 3;
    IF (EnvironmentContext.Activity >= on_foot)
    THEN (
        SwitchUIModality(BookDetailsView, vocal);
    );
}
```

Fig. 4.3 Example: Integrated Modeling of Self-adaptive UIs

and context model, *AdaptML* allows to specify and bind different adaptation rules to the *IFML* model. The UI adaptation model, depicted on the bottom right side of Figure 4.3, contains example UI adaptation rules. The first adaptation rule specifies a *TaskChangeOperation* based on the user role. In this case, it is checked whether the user has the role *admin*. If this is the case, an additional model element can be added by using the *AddIFMLElementOperation* which adds a further *adminEvent* element to the *bookDetails* component in the example. More details about the available adaptation operations are provided in Section 4.4. The second adaptation rule is called *NavigationChangeOperation based on UserRole* and defines that the specific view *BookDetailsView* can be only reached, if a specific user context is satisfied. For defining this adaptation behavior, *AdaptML* rules are referencing the context model where relevant contextual parameters are described and the *IFML* model to reference the specific UI model elements that has to be changed. In the case of our example, the user role *user* has to be satisfied, so that the *BookDetailsView* can be reached. In a similar way, various other UI adaptation rules can be specified. Analogously, the third rule is defining a *LayoutChangeOperation* to increase the font size if the user's vision is under the specified threshold value. Finally, the last adaptation rule specifies a *ModalityChangeOperation* for the *BookDetailsView* which switches its UI modality from graphical to vocal if a movement is detected.

4.3 Context Modeling with *ContextML*

In this section, we present our novel context modeling language *ContextML* in more detail. Context management and especially the specification of contextual parameters was identified as one of the essential prerequisites to support self-adaptive UIs. To support a holistic modeling of contextual parameters and address the challenge *C1: Specification of contextual parameters* conform to the described language engineering requirements, we developed the modeling language *ContextML*. It enables to define a set of context properties and the needed context provider interfaces to capture the relevant context information. These interfaces are later on referenced as *Context Providers*. For example, the environmental light condition of a context-of-use can be captured by using a provided *AmbientLightAPI*. Also the types of these properties and their behavior, in terms of data updating, has to be defined. For instance, ambient light is monitored as current light-level, and it shall be updated as soon as a change is detected (event-triggered). Figure 4.4 depicts an overview of *ContextML*'s metamodel.

The root element and central class of the metamodel is the class *ContextML* that connects all parts of the metamodel. The root class *ContextML* is further specialized by specific context entities, such as *UserContext*, *PlatformContext*, *EnvironmentContext*, and *Custom-Context*. Each context entity *UserContext*, *PlatformContext*, and *EnvironmentContext* has a *ContextProperty* (abstract class) which covers *name* and *updateVelocity* of a contextual parameter. The abstract class *ContextProperty* has been introduced to ease the reference from *AdaptML* through a central class which connects all the needed context entities. Moreover, the reference to the desired *ContextProvider* is stored, which is the source of context information and is provided through a context sensor. The data type of a *ContextProperty* provided through a *ContextProvider* can be a standard type like *Integer*, *String* or *Boolean*, but also a user-defined type is supported. The *updateVelocity* describes the way, how a single data set shall be updated. This can be *"slow"*, *"fast"*, or *"eventTriggered"*, which is whenever a context information change occurs.

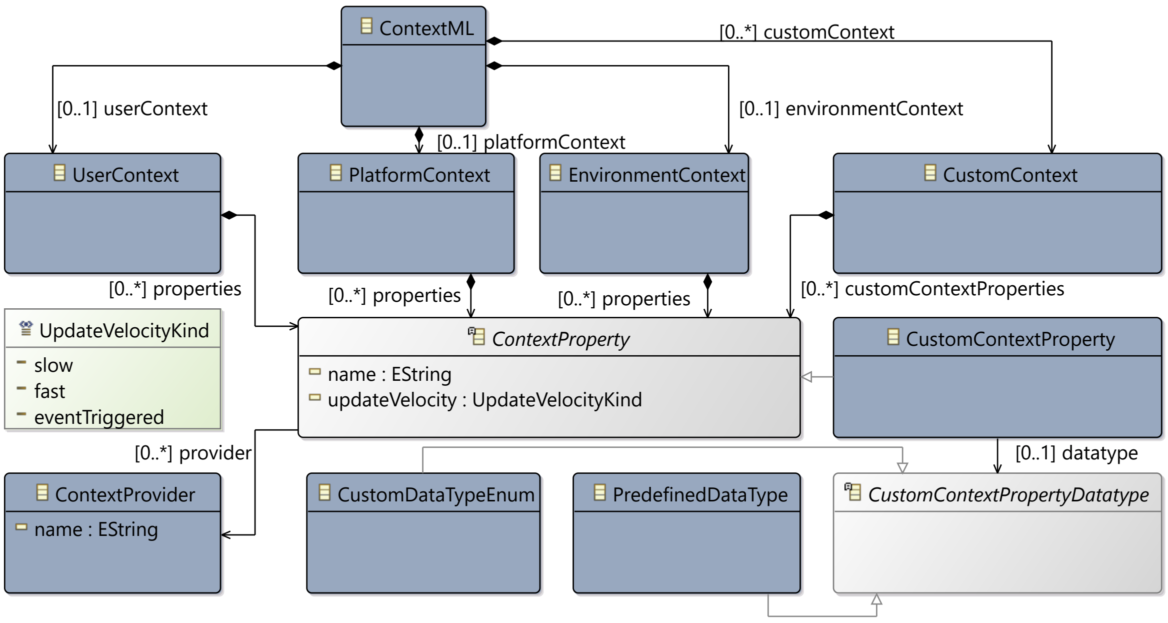


Fig. 4.4 *ContextML*: Overview of the Context Metamodel

For supporting the language requirements *Extensibility* and *Genericity*, the metamodel is designed for a broad scope of context modeling aspects and allows adding further context entities via *CustomContext*. Each *CustomContext* can have a *CustomContextProperty* inherited by *ContextProperty*. Moreover, each *CustomContextProperty* has a *CustomContextPropertyDatatype* which can be a *CustomDataTypeEnum* or a *PredefinedDataType*. In order to support a broad spectrum of contextual parameters and ease the work of the developers in specifying various context-of-use situations, *ContextML* comes up with a fine-grained modeling support to cover the context triplet *UserContext*, *PlatformContext*, and *EnvironmentContext*. In the following, those context categories are described in more detail. It should be noticed that each context entity *UserContext*, *PlatformContext*, and *EnvironmentContext* is described through explicit context property classes, in most cases denoted with a [0..1] cardinality. Beside implementation specific reasons (support better auto-completion support in the modeling workbench) we have chosen this metamodel structure in order to provide a flexible specification of the relevant context properties. Figure 4.5 depicts a refinement of *UserContext* as part of the *ContextML* metamodel. It covers several context information related to the actual user and each user has a unique *UserID*. Typical context properties about the user are age, gender, and language. The context property age can be specified as *AgeUserDefined* when it is gathered through a user prompt or *AgeCalculated* when it is detected through a camera and estimated through a machine learning algorithm. Beside that, the *Mood* of the user (happy, angry, suprised, etc.) and if she/he wears *Glasses* and has a

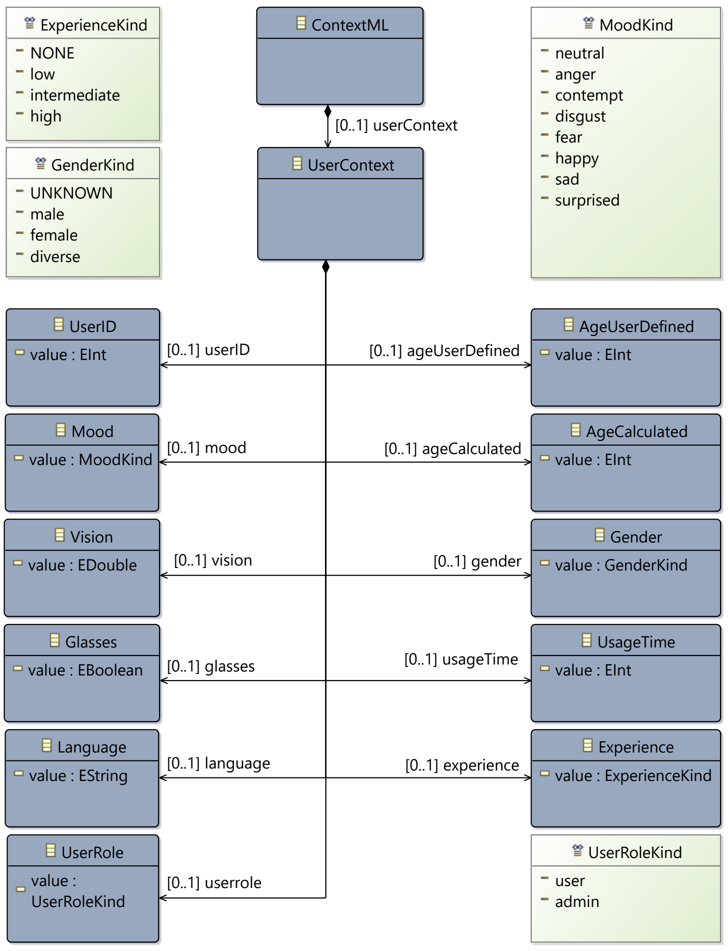


Fig. 4.5 *ContextML* metamodel for *UserContext*

Vision problem can be also detected through a camera and face detection API. Furthermore, when a user model is specified, we can ask the users for context information about their *Experience* level or *UserRole* for interacting with the system. Lastly, we can specify and track the *UsageTime* of each user.

In a similar way, *PlatformContext* (see Figure 4.6) entity covers aspects according to the execution platform that should be considered when maintaining usability of the UI. As with *UserContext*, each *PlatformContext* has a unique *PlatformID* so that the platform model is mapped to the correct execution platform. Common context properties regarding execution platform are *OSName* and *OSVersion* denoting the name of the running operating system and its version number, respectively. In addition to that, the context property *DeviceType* characterizes the device type of the execution platform, such as desktop, mobile or tablet. Also, the specific *DeviceName* and *TimeZone* can be specified and observed as an additional context information. A pair of context properties that is essential for the UI is the screen

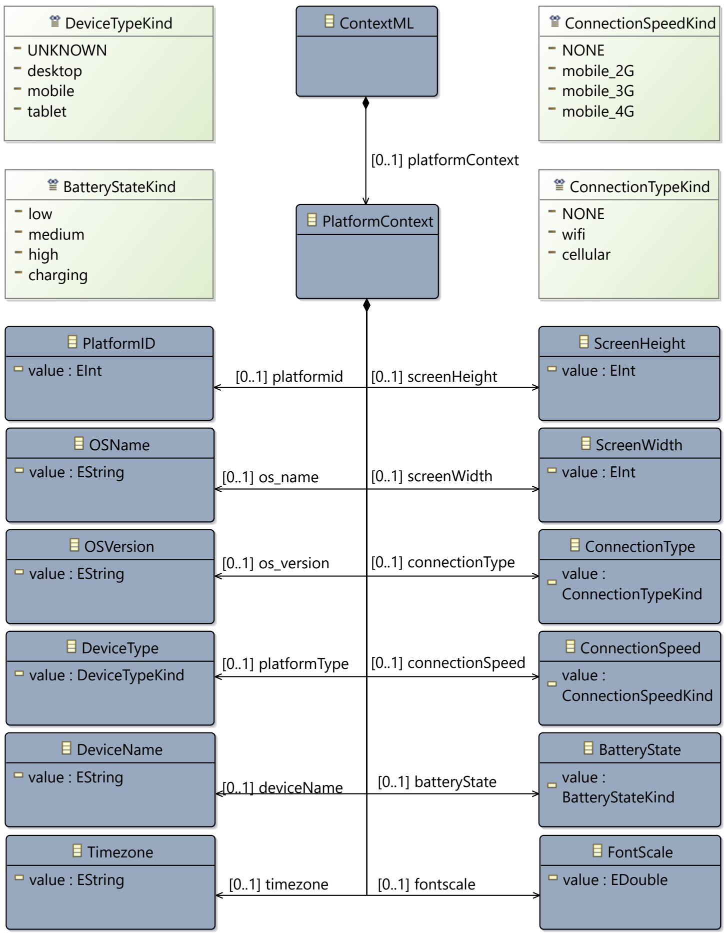


Fig. 4.6 *ContextML* metamodel for *PlatformContext*

dimension. In our metamodel it is represented by *ScreenHeight* and *ScreenWidth* for the respective dimension height and width. These context properties are often used to scale the UI to the respective device. The most important scaling property is that for font, because UIs mostly contain text elements. As text elements play a major role for UIs, a *FontScale* is introduced. It carries the default font scale for the respective device to take into consideration when modifying its value. Furthermore, dynamic platform context properties such as *ConnectionType* (wifi, cellular, etc.) and *ConnectionSpeed* (mobile_2G, mobile_3G, etc.) can be used to track the quality and speed of the internet connection. Finally, the dynamic platform context property *BatteryState* is an important context information that can influence functionality and usability of the UI.

The *EnvironmentContext* part of the *ContextML* metamodel, depicted in Figure 4.7, is especially important when considering the mobile scenario. In the mobile scenario, the context parameters regarding the interaction environment can dynamically change. Several dynamic context changes and various combinations of environmental context properties

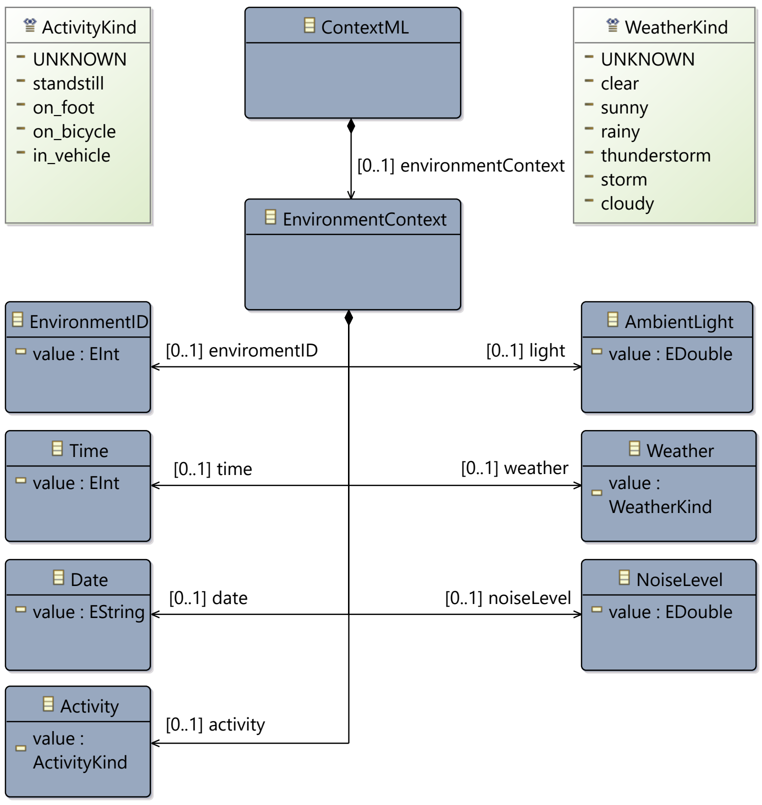
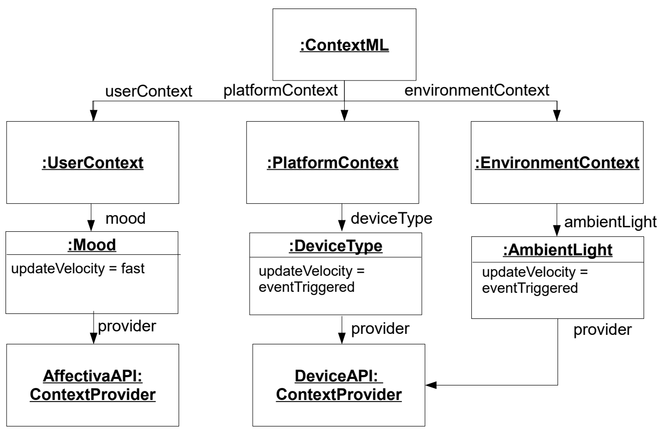


Fig. 4.7 *ContextML* metamodel for *EnvironmentContext*

are conceivable. In *ContextML*, the most relevant environmental properties are considered. These are *Date*, *Time*, *AmbientLight*, *NoiseLevel*, *Weather*, and *Activity*. While *Date* and *Time* constitute the temporal dimension, *Light*, *NoiseLevel*, *Weather*, and *Activity* constitute the space dimension. As part of the space related context properties *Activity* is defined by an enumeration over different states of movement such as *standstill*, *on_foot*, *on_bicycle*, etc. Similarly, context information about the *Weather* are defined through an enumeration over different states (clear, sunny, rainy, etc.).

An excerpt of an example context model based on *ContextML* is depicted in Figure 4.8. It shows a set of possible context entities. For illustrating the context modeling language, example entities which contain some illustrative context properties are shown. The example context model covers three different context entities *UserContext*, *PlatformContext*, and *EnvironmentContext*. Those context entities are specifically described through context properties. For example the context entity *UserContext* has a context properties *Mood* that is characterized through suitable data and update types. Also the the context provider as a source of the context property information is described which in this case is the *AffectivaAPI* to get the mood information about the user. Based on the metamodel, we also created a



```
ContextML{
    Enitity UserContext {
        Mood {
            provider: AffectivaAPI;
            updateKind: fast;
        };
    }
    Enitity PlatformContext {
        DeviceType {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
    }
    Enitity EnvironmentContext {
        AmbientLight {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
    }
    Providers{
        DeviceAPI;
        AffectivaAPI;
    }
}
```

Fig. 4.8 Example *ContextML* model: graphical (left) and textual concrete syntax (right)

concrete syntax for *ContextML* using *Xtext*[1] (see Section 2.6.3). Based on *Xtext*, we created an Eclipse plugin for context modeling which allows an easy modeling of the context, due to error highlighting and code completion (see Section 6.3 Tool-Support for further details). This way, the required programming knowledge and error potential is reduced. An example of the concrete syntax is displayed on the right side of Figure 4.8.

In summary, the context model specified in *ContextML* serves as an input for the *Context Service Generator*. The latter creates executable code for the *Context Service* with a Model-to-Text-Transformation (M2T). The generation of executable *Context Services* in the form of code is described in the next chapter.

4.4 Adaptation Modeling with *AdaptML*

In the previous section, we introduced *ContextML* as a generic and extensible modeling language for specifying various heterogeneous context-of-use situations. Beside characterizing the different context-of-use parameters that can influence the usability of the operated UI, UI developers have to be supported in the task of specifying various UI adaptations. In order to support the UI adaptation modeling approach and address challenge *C4: Specification of UI adaptation rules* conform to the described language engineering requirements, we present *AdaptML*. The main purpose of *AdaptML* is to provide a dedicated modeling perspective (Separation of Concerns) which allows the separate specification of UI adaptation rules complementary to the UI and context model (Integration). *AdaptML* aims to support the

[1]https://eclipse.org/Xtext/

specification of various UI adaptations by covering different adaptation techniques and reduce complexity in designing and maintaining adaptation rules.

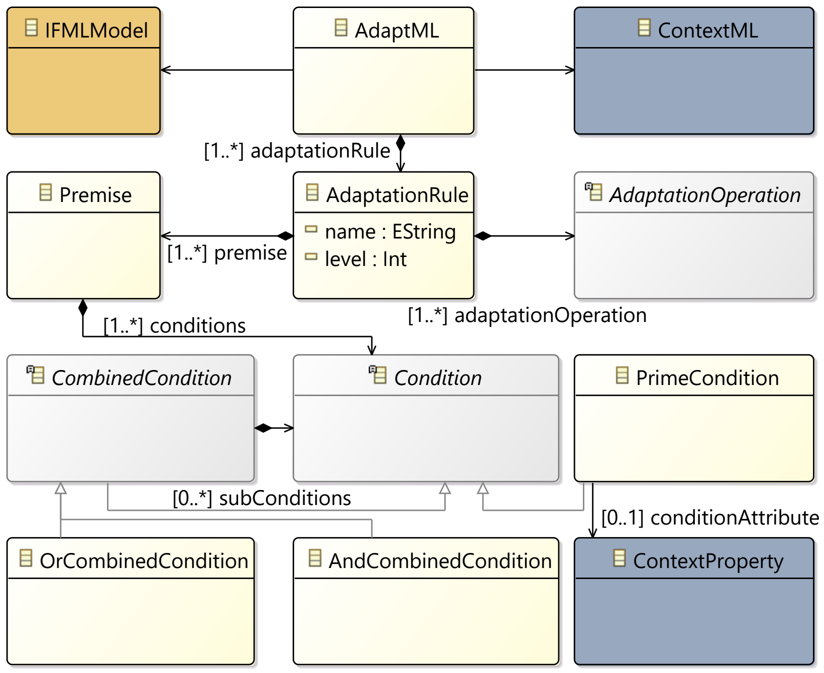


Fig. 4.9 *AdaptML*: Adaptation Metamodel Overview

An overview of the general structure of the *AdaptML* language is shown in Figure 4.9. The root element of the metamodel is the *AdaptML* class. In order to achieve the mentioned language engineering requirement *Integration*, the *AdaptML* class has a reference to the *ContextML* class to evaluate the context conditions and a reference to the *IFMLModel* class to define UI adaptations for specific UI elements. *AdaptML* consists of *AdaptationRule* elements which have a rule *name* and a priority *level* as attributes. The priority *level* is used as an indicator for priority to decide in which order rules are executed if more than one satisfies all conditions. A rule with higher priority level is executed before rules with lower level. Each *AdaptationRule* consists of one or more *Premise* and *AdaptationOperation* elements. The *Premise* class characterizes the condition part of a UI adaptation and consists of an abstract class *Condition* which is the base for describing simple and complex conditions. Simple conditions can be specified based on the *PrimeCondition* class which defines a concrete simple constraint on one *ContextProperty*. Therefore, *PrimeCondition* has the attributes *operator* and *value* which are needed to define a condition of a UI *AdaptationRule* expressed as a logical expression.

More complex conditions can be specified through the abstract class *CombinedCondition* which allows a combination of conditional expressions concatenated by OR-operators and

AND-operators. For this purpose, the subclasses *OrCombinedCondition* and *AndCombined-Condition* are defined in the *AdaptML* metamodel.

Beside the above described conditional expressions, an *AdaptationRule* enables to specify different UI adaptation techniques that are executed if the associated conditions are satisfied at runtime. For this purpose, each *AdaptationRule* consists of one or many *AdaptationOperation* elements. In order to fulfill the concern-specific requirements introduced in Section 3.2, the *AdaptML* language supports different types of adaptation techniques which are depicted in Figure 4.10: *TaskChangeOperation*, *NavigationChangeOperation*, *LayoutChangeOperation*, *ModalityChangeOperation*, and *ServiceOperation*. Also, a combination of multiple adaptation techniques is possible. In *AdaptML*, this is implicitly modelled by the composition relation between *AdaptationRule* and *AdaptationOperation*.

TaskChangeOperation enables the specification of UI adaptations which allow to decrease and increase the task-feature-set to provide a more minimalistic or detailed UI view upon the current context-of-use. For hiding and showing specific UI elements, *TaskChangeOperation* supports the *AddIFMLElementOperation* and *DeleteIFMLElementOperation* through a *targetPath* which enables to apply these operations on specific UI elements of the specified *IFML* model. For this purpose, *TaskChangeOperation* has a target reference to the *IFML* class *InteractionFlowElement*.

Similarly, *NavigationChangeOperation* enables to specify UI adaptations where the navigation flow of a UI can be changed based on the context-of-use. For this purpose, *NavigationChangeOperation* has the subclasses *AddNavLinkOperation*, *DeleteNavLinkOperation*, *RedirectNavLinkOperation*, and *ClearNavOperation*. The first three classes or navigation change operations support the addition, deletion, and redirection of a specific navigation edge, denoted as *NavigationFlow* in the *IFML* model, which is referenced through a *targetPath*. Lastly, the navigation change operation *ClearNavOperation* can be used to remove all links that are currently stored in the navigation component.

As a further UI adaptation technique, *AdaptML* supports the specification of layout changes which are characterized through the *LayoutChangeOperation* class. Although *IFML* in general is an abstract UI modeling language which is not directly focusing on platform-specific details like layout, we decided to incorporate layout change operations in *AdaptML* as we see a higher potential and flexibility for UI adaptations through this possibility. The *LayoutChangeOperation(s)* are mainly inspired by commonly used Cascading Style Sheet (CSS)[2] properties. The main idea is to ease the developers work in specifying layout change operations by reusing a common standard and integrating it into our modeling approach as it is also the case with *IFML*. Similar to the *targetPath* attribute of the *TaskChangeOperation(s)*,

[2]https://www.w3.org/Style/CSS/Overview.en.html

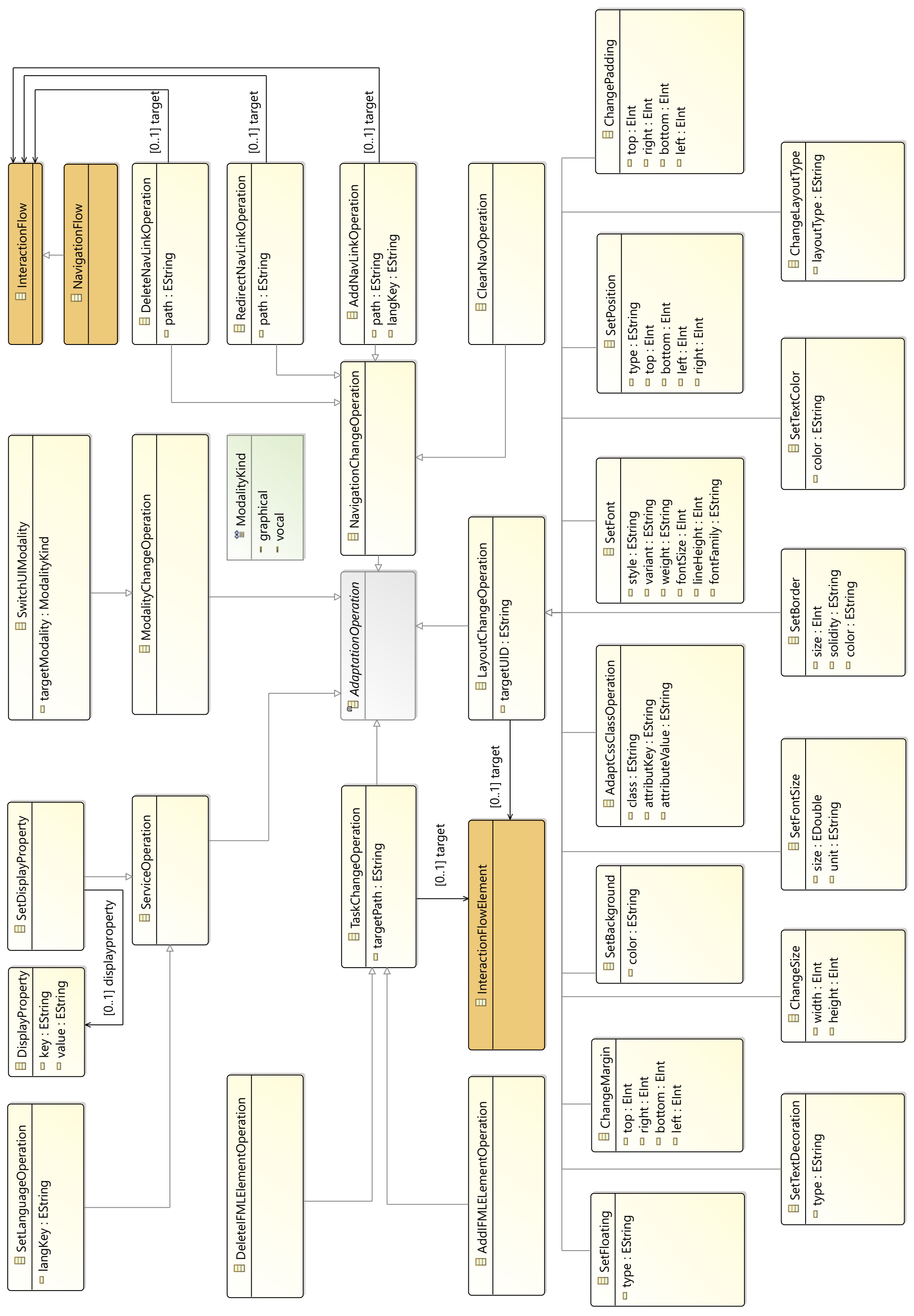


Fig. 4.10 *AdaptML*: Overview Adaptation Operations

the *LayoutChangeOperation* class contains the *targetUID* attribute identifying a UI element uniquely. This way, different *LayoutChangeOperation(s)* can be applied to specific UI elements which are contained in the *IFML* class *InteractionFlowElement*. Typical layout change operations commonly known from CSS are for example *SetFontSize*, *SetPosition* or *SetTextColor*. Furthermore, we added the *ChangeLayoutType* operation to support a switch between different layout types like grid or linear layout. Finally, we also added a generic *AdaptCSSClassOperation*. With this operation it is possible to set fine granular style properties for a specific class of UI elements. The *class* attribute in *AdaptCSSClassOperation* is equal to a CSS class, while the *attributeKey* is intended to be a CSS property and the *attributeValue* a valid value to that key. The *AdaptCSSClassOperation* is not intended to be used in general, but represents an alternative solution in cases where the specific *LayoutChangeOperation* is not implemented in *AdaptML*.

In addition to *LayoutChangeOperation*, *AdaptML* also supports the specification of basic modality changes for the interaction with the UI. For this purpose, the *ModalityChange-Operation* class is provided which enables to switch the interaction modality type via the *SwitchUIModality* class between graphical and vocal user interface.

As a last type of adaptation technique *AdaptML* supports a *ServiceOperation* in the target language of the UI. In our approach, reusable predefined Angular services were provided to support UI adaptations for the web platform. The definition of these services enables to use them later on in the rule specification. A *ServiceOperation* is defined by its name and relative location to the *Services* folder of the Angular implementation. A *ServiceOperation* can contain interfaces to functions. *ServiceOperation(s)* are helpful to specify UI changes that affect bigger parts or a group of UI elements. For this purpose, *ServiceOperation* has a subclass *SetDisplayProperty* which takes a *DisplayProperty* object as input and enables a group of changes in the UI. As an example, *AdaptML* comes up with a predefined *SetLanguageOperation* which supports internationalization of the UI language. Therefore, each text on the UI is represented by a language key which is automatically set to the detected language on the used device.

Based on *Xtext* (see Section 2.6.3), we also created an Eclipse plugin for adaptation modeling which allows an easy modeling of UI adaptation rules, due to error highlighting and code completion (see Section 6.3 Tool-Support for further details). This way, the required programming knowledge and error potential is reduced. An example of the concrete syntax for *AdaptML* is shown in Figure 4.11, which contains example UI adaptation rules.

The first UI adaptation rule specifies a *TaskChangeOperation* based on user's mood. If a *sad* mood is detected through the face detection API, a *helpEvent* is added through the *AddIFMLElementOperation* to provide for instance a textual help for the user. In the second

```
rule "TaskChangeOperation based on User's Mood"{
    LEVEL 1;
    IF(UserContext.Mood == sad) THEN
    (
        AddIFMLElementOperation(BookDetailsView, helpEvent);
    );
}
rule "LayoutChangeOperation based on Environment's Light (0-100 %)"{
    LEVEL 2;
    IF(EnvironmentContext.AmbientLight < 100) THEN
    (
        AdaptCssClassOperation("ViewContainer","filter","contrast(0.5)");
    );
}
rule "LayoutChangeOperation based on User's Experience"{
    LEVEL 1;
    IF(UserContext.Experience <= low ) THEN
    (
        ChangeLayoutType("sidebar-navbar", grid);
    );
}
rule "ModalityChangeOperation based Environment's Activity"{
    LEVEL 3;
    IF(EnvironmentContext.Activity >= on_foot ) THEN
    (
        SwitchUIModality(BookDetailsView, vocal);
    );
}
rule "ServiceOperation based on User's Language"{
    LEVEL 1;
    IF(UserContext.Language === de ) THEN(
        SetLanguageOperation("dede");
    );
}
```

Fig. 4.11 Example UI adaptation rules based on *AdaptML*

adaptation rule a *LayoutChangeOperation* based on the environmental light condition is specified. When the light condition is under a certain threshold value, the contrast of the UI is increased through the *AdaptCSSClassOperation*. Similarly, the third adaptation rule specifies a *LayoutChangeOperation* based on the user's experience level while the fourth rule encodes a *ModalityChangeOperation* to trigger a switch in to the vocal UI if a movement is detected. Finally, a language adaptation is shown in the last adaptation rule where the UI language is switched based on the user's currently used language using the *ServiceOperation*.

In summary, *AdaptML* models serve as an input for the *Adaptation Service Generator*. The latter generates the adaptation logic for the *Adaptation Service* with a Model-to-Text-Transformation (M2T). The generation of executable Adaptation Services in the form of code is described in the next chapter.

4.5 Summary and Discussion

In this chapter, an integrated modeling framework for self-adaptive UIs has been introduced. The modeling framework is based on OMG's standard UI modeling language *IFML* and builds around two complementary novel modeling languages *ContextML* and *AdaptML*. Through integration of these modeling languages, we allow a comprehensive specification of self-adaptive UIs. In the following, we recapture the high-level and concern-specific requirements considering our presented modeling framework.

- *Separation of Concerns:* Through the integration of *IFML*, *ContextML*, and *AdaptML* we provide different modeling views which allow to focus on the relevant concerns such as UI, context, and adaptation. This way, developers are able to specify the relevant concerns in an appropriate and separate way. The relevant core concerns are also separated from the system's business logic in order to reduce the complexity.
- *Integration:* As already mentioned above, our modeling framework is designed in a way that it is fully integrated with OMG's standard UI modeling language *IFML*. Therefore, similar development projects which aim to devise self-adaptive UIs and rely on *IFML* can use our modeling framework. As *IFML* is an OMG standard it complies with other OMG standardized languages such as *UML* or *BPMN* which can be also seen as an advantage for our modeling framework.
- *Intuitiveness:* Our modeling framework makes use of standard modeling techniques such as UML class diagrams for the domain model and *IFML* for the abstract UI model. Also the newly introduced domain specific languages *ContextML* and *AdaptML* are smoothly integrated in our modeling workbench (see Section 6.3) allowing clear separation of different internal concerns which help to master the complexity of modeling self-adaptive UIs.
- *Extensibility:* Both newly introduced modeling languages *ContextML* and *AdaptML* are designed in a way that new modeling concepts can be flexibly added. In *ContextML*, for this purpose, we have introduced the concept of *CustomContextPoperty* which allows developers to define their own context-of-use parameters based on suitable self-defined data types. In *AdaptML* extensibility is supported through the *AdaptationOperation* class which can be extended to further UI adaptation techniques.
- *Genericity:* Our modeling framework is designed with genericity in mind. That is, our modeling framework provides concepts and techniques to specify self-adaptive UIs for different application domains.

Regarding the concern-specific requirements according to context and UI adaptation, we can sum up that all relevant concerns are addressed by our modeling framework. The context aspects user, platform, and environment are covered in detail by *ContextML* and also *AdaptML* provides means to define various UI adaptations based on different adaptation techniques such as *TaskChangeOperation*, *NavigationChangeOperation* or *ModalityChangeOperation*.

Chapter 5

Transformation

This chapter describes our transformation approach for self-adaptive user interfaces. Firstly, in Section 5.1, we provide an overview of the general architecture for our code generator for self-adaptive UIs. In Section 5.2, we describe the structure and functionality of the *UI Generator* which is responsible for automatically creating the final user interface. Analogously, Section 5.3 serves to present the *Context Service Generator* which is responsible for automatically deriving a *Context Service* that monitors the contextual parameter. Section 5.4 deals with the generation of an *Adaptation Service*. Finally, Section 5.5 concludes this chapter with a brief summary and discussion.

5.1 Transformation Approach Overview

In the previous chapter, we have presented our modeling framework for self-adaptive UIs. In order to support the utilization of our modeling framework and the model-driven development approach for devising self-adaptive UIs, we implemented a code generator for self-adaptive UIs (*SAUI-Generator*). Our code generation approach is targeting web applications as they are widely spread and used in various application domains. Relying on the principles of the World Wide Web (WWWW[1]), web applications provide many advantages (e.g., portability interoperability, accessibility, etc.). With this regard, one of the most important advantage of web applications is that they are accessible via a browser from different target platforms such as mobile, tablet or desktop. Thus, in order to use the potential of web applications as an enabler for self-adaptive UIs, our code generation approach is focusing on web applications as target technology. As already described in Section 2.6, *Angular* represents a de facto standard framework for developing web applications. Therefore, our code generation approach is

[1]https://www.w3.org/

especially based on and illustrated by focusing on this target technology. At this point, it should be also noticed that our code generation approach mainly focuses on the core aspects of self-adaptive UIs, which are UI, context, and adaptation whereas business logic, data management and other related aspects of web applications are out of scope.

Figure 5.1 shows the overall architecture of the implemented *SAUI-Generator*. It consists of a main generator, *Generator Core*, and three sub-generators *UI Generator*, *Context Service Generator*, and *Adaptation Service Generator*.

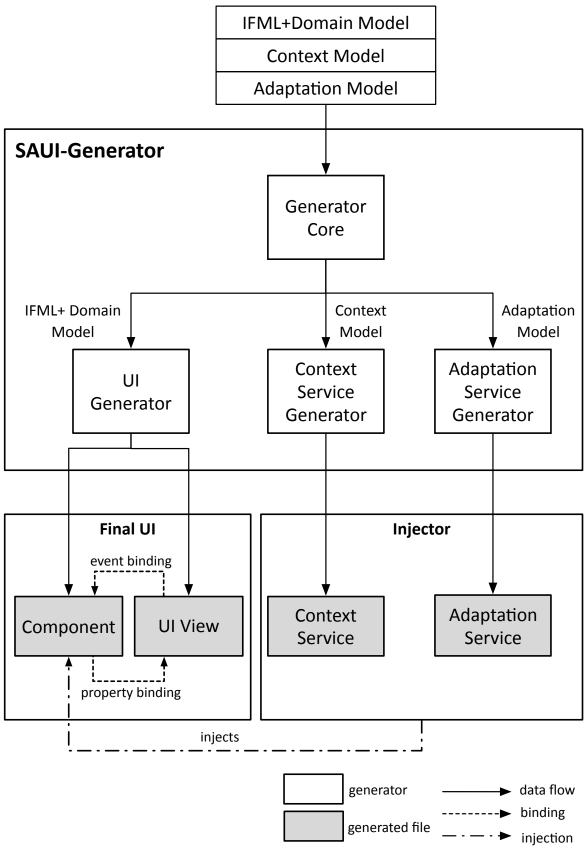


Fig. 5.1 Architecture Overview of the SAUI-Generator

The *Generator Core* gets as input the *IFML* and *Domain Model* as well as the *Context* and *Adaptation Model* which were specified based on the presented integrated modeling approach in the previous chapter. The models are then delegated to the corresponding sub-generators. Based on the *IFML* and *Domain Model*, the *UI Generator* automatically creates the *Final UI* as Angular *Components* and *UI Views*. Furthermore, based on the specified *Context* and *Adaptation Model*, the *Context* and *Adaptation Services* are generated, respectively. The generated services are injected into the *Component* element of the *Final UI* by using the Angular *Injector*[2] technique (see Section 2.6.1 for further details).

The main idea of the generation approach relies on Model-to-Text (M2T) Transformations based on Xtend as transformation language (see Section 2.6.3). In the following sections, we describe the implementation of the specific code generators responsible for the automated creation of the *UI, Context*, and *Adaptation Services*. The interplay of those generated elements within the execution environment is essential to support context management and UI adaptation at runtime.

5.2 UI Generation

The main goal of the *UI Generator* is to automatically create the *Final UI* (FUI) in the means of Angular *UI Views* (see Section 2.6.1). The *UI Views* in Angular are mainly represented by two parts, a *Template* and a *Component* (see parts of Figure 5.2 marked blue). A *Component* defines a graphical view that contains display and interaction elements like a list or buttons. An application's UI is a hierarchy of multiple components starting from a root component. The *Component* element usually also contains the application logic which can be injected through *Services* characterizing specific function calls. In our case, this technique is used to integrate the *Context* and *Adaptation Services* which is described later on in Section 6.2. The *Template* of a *Component* is defined as HTML markup which is extended by an Angular specific notation, called *Directive*, to achieve a property binding and event binding between the *Component* and *Template* in order to access data and trigger interaction events.

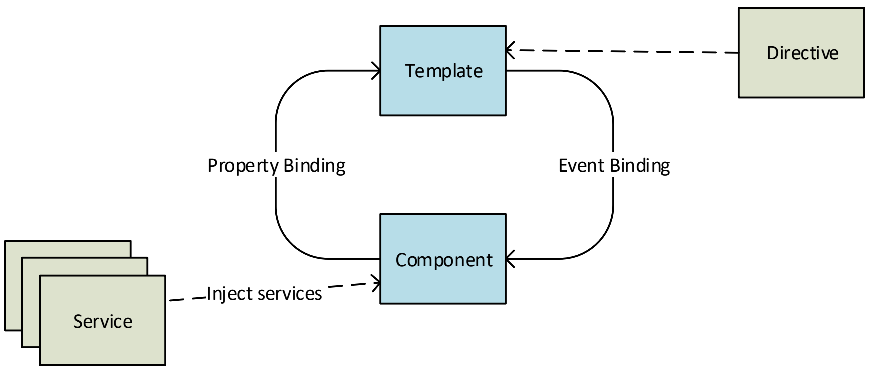


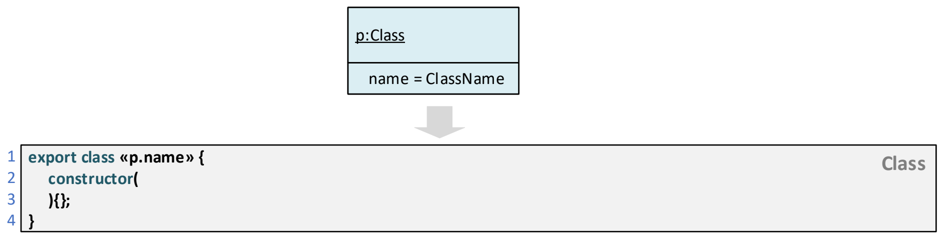
Fig. 5.2 Simplified Overview: Angular Framework (UI view parts marked blue) [GS13]

[2]https://angular.io/api/core/Injector

As decribed in the previous chapter, the UI is modeled by an *IFML* model and a domain model represented as a *UML* class diagram. The elements of these models need to be mapped into Angular *UI Views*. To achieve this, model-to-text transformation templates for the elements of these models have been defined. In the following, firstly, the Xtend templates for transformation of the domain models to *Angular Classes* are presented, then a selection of Xtend templates for the transformation of the *IFML* model elements to corresponding *Angular* target elements are described.

5.2.1 Mapping: *DomainModel2AngularClasses*

The input domain model is represented as a *UML* class diagram. The specific UML Classes in the UML class diagram need to be represented as *Angular* classes in the target file. The mappings are visualized in the following figures. The currently selected source model element for the mapping is marked blue and shown on the top of the grey mapping arrow. Below this, the newly created elements of the target artifacts are shown. As a general rule for source (top) and target (below), elements with white background define the application context of the rule.

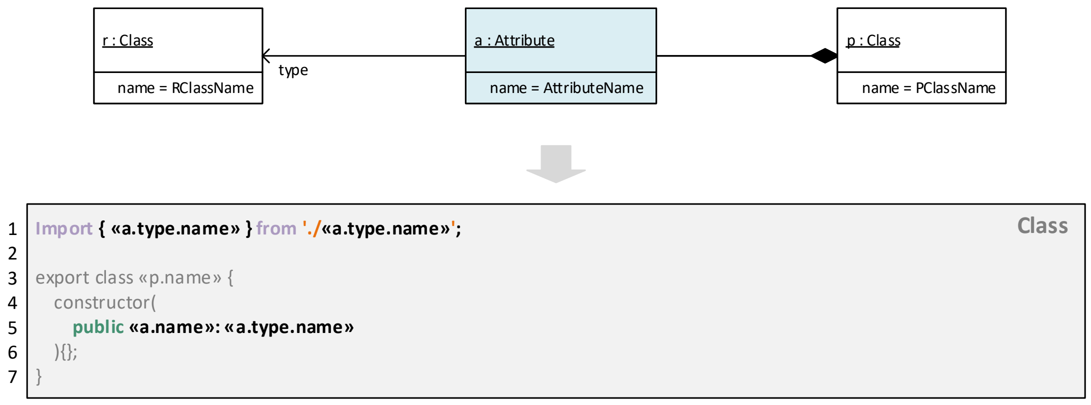


```
export class «p.name» {
   constructor(
   ){};
}
```

Fig. 5.3 *UMLClass2AngularClass* mapping

For each *UML Class*, a new *Angular* class file, implemented in TypeScript, will be created with the *name* of the UML class as the Angular Class *name* (see Fig 5.3). The class content initially only consists of an empty constructor.

Within the Angular class, the variable declarations for the UML attributes have to be created. There are two possible mappings which depend on the type of the attribute: primitive and complex types. In both cases, the declaration of a variable is a parameter of the constructor. Then, TypeScript automatically creates the global declaration when it is compiled to JavaScript at compile time of the UI. For variables of a primitive type, like a boolean value, just the variable name and its type are created as declaration in the constructor parameter list. A global variable is created when the TypeScript code is compiled to JavaScript. If an unknown type is used, the variable is declared as being of the type ’any’.



```
Import { «a.type.name» } from './«a.type.name»';

export class «p.name» {
  constructor(
    public «a.name»: «a.type.name»
  ){};
}
```

Fig. 5.4 *UMLClass2AngularClass*: Attribute mapping for primitive types

If the type of the attribute refers to another class in the domain model, additionally to the variable declaration in the constructor, the other class needs to be imported into the Angular class (see Fig 5.4). The import statement is added in the header of the Angular class. The variable declaration as parameter of the constructor is the same as for primitive types. The location of the imported class is known, since it is also created by the generator.

5.2.2 Mapping: *IFML2AngularViews*

Similar as the domain model, the abstract UI model, represented in *IFML*, needs to be mapped to Angular specific target elements to cover the *Final UI*. For this purpose, a subset of *IFML* model elements has been selected and mapped to corresponding *Angular Elements*[3] to establish the concept of the UI generation part of this thesis. Our transformation approach for UI generation supports the mappings that are listed in Table 5.1.

The source elements in Table 5.1 describe *IFML* model elements that were introduced in Section 2.2.2, while the targeting elements on the right column refer to specific *Angular Elements* to represent the abstract UI model elements through concrete web interaction elements. In general, the source model elements can be divided into two groups: simple elements without child elements and complex model elements which can contain other nested *IFML* model elements as child elements. In the following, we describe the *IFML2Angular* mapping relations based on illustrative simple and complex elements.

[3]https://angular.io/guide/elements

Source Element	**Target Element**
Action	*Function stub in Angular Component*
Data Binding	*HTML content binding and Angular property binding*
Data Flow	*Function call*
Details	*HTML table for one element*
Form	*HTML form*
List	*HTML table for multiple elements*
Navigation Flow	*Navigation to an Angular View*
OnSelectEvent	*HTML button and function stub*
OnSubmitEvent	*HTML button and function stub*
Parameter	*Selected element of HTML table and Angular property binding*
Parameter Binding	*URL parameter*
Parameter Binding Group	*URL parameters*
Simple Field	*Field in HTML form*
View Container	*Angular Component + Template*
Visualization Attribute	*Column in HTML list*

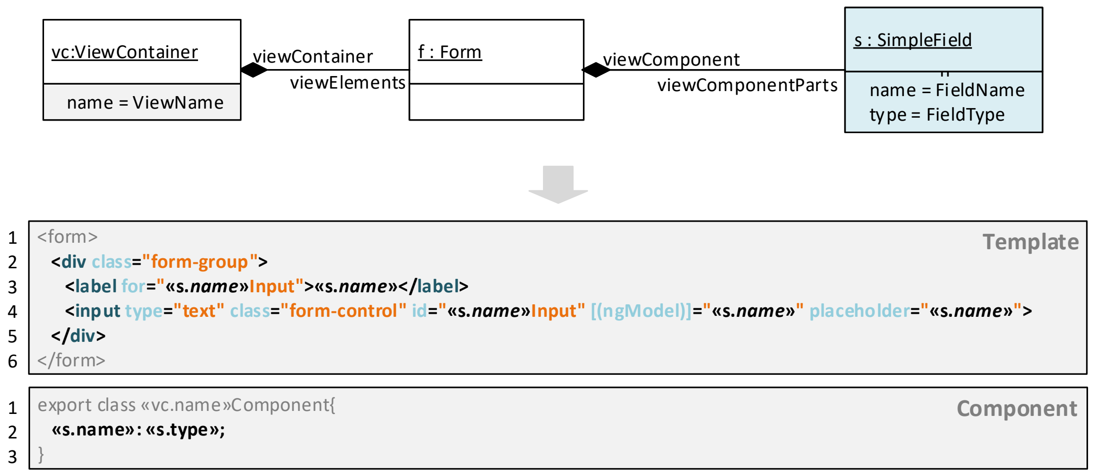
Table 5.1 *IFML2AngularElement* Mappings for UI Generation

Simple elements

Simple elements in context of the mapping are defined as *IFML* model elements that are not a parent container of other elements. Since the target technological environment of the transformation is the Angular framework, the mappings are defined with this in mind. As described before, a view in Angular consists of a *Template* and a *Component*. The *Template* is the HTML representation of the view, while the *Component* is the view controller, implemented in TypeScript. To visualize the mappings, an excerpt of the *IFML* model is shown. Elements with blue background in the *IFML* model are the current focus of the mapping. Elements with white background are the context of the mapping. In addition to the *IFML* model elements, the representation of those elements in Angular is shown. The Angular representation is split in *Template*, for the HTML representation, and *Component*, for the TypeScript view controller. Colored code excerpts are the part of the code that is newly created. Code parts that are grey are indicating the context of the excerpt. References to attributes of the *IFML* model are indicated with the ≪≫-brackets.

Source Element: *SimpleField*
Target Element: HTML Input + Label, Variable for property binding
Chronology: *SimpleFields* are child elements of *IFML Forms*. This means, the transformation for *SimpleFields* happens within the transformation for *Forms*. However, the property

binding, as it is also part of the Angular *Component*, is generated during the *ViewContainer* generation which is described later in this Section (see Figure 5.10).



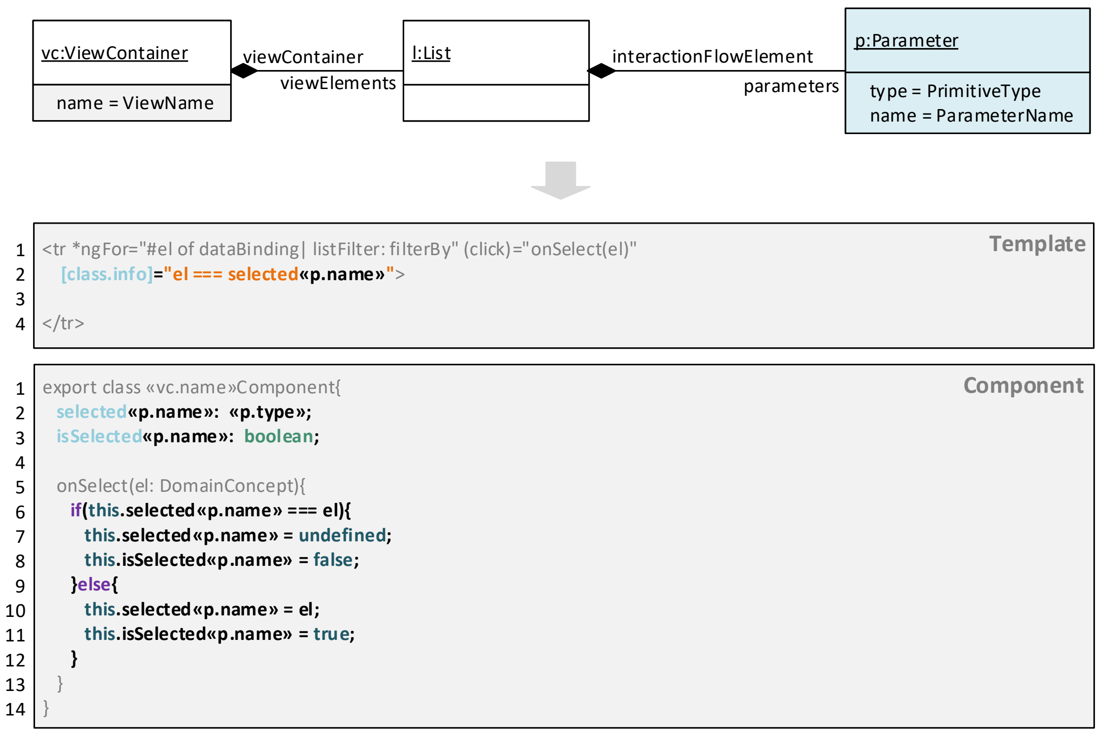
```
1 <form>
2   <div class="form-group">
3     <label for="«s.name»Input">«s.name»</label>
4     <input type="text" class="form-control" id="«s.name»Input" [(ngModel)]="«s.name»" placeholder="«s.name»">
5   </div>
6 </form>
```

```
1 export class «vc.name»Component{
2   «s.name»: «s.type»;
3 }
```

Fig. 5.5 Mapping for *SimpleField* element

Description: *SimpleFields* have a type, indicating the data type of the field, and a name associated with them (see Figure 5.5). Every *SimpleField* is associated with an *IFML Form*. Likewise, in the Angular *Template* they are created as child elements of an HTML form (*Template* Line 2-5). Every *SimpleField* is represented by a *label*-element and an *input*-element. The label is the text displayed next to the input, which is a simple text input, to give the user an idea of the information the field should contain. In the mapping, there is already some layout information, in form of CSS classes ('form-group' and 'form-control'), present. In the input-element, there is an *ngModel*-attribute. This attribute is responsible for establishing the property binding with the Angular variable (*Component* Line 2) for the field. This means, when information is added to the field, it needs to be shared with the *Component*. Likewise, if information changes occur in the *Component*, due to the bidirectional binding, these changes are reflected to the input field.

Source Element: *Parameter*
Target Element: Selected data item (e.g., row in HTML table), Variables for Property Binding
Chronology: In Figure 5.6, the mapping for a parameter associated to a list is shown. The transformation of parameters with other associated contexts is similar. However, during the generation of the list, which is represented by an HTML list, an attribute for conditionally changing the CSS class of the table row is added.



```
1  <tr *ngFor="#el of dataBinding| listFilter: filterBy" (click)="onSelect(el)"
2    [class.info]="el === selected«p.name»">
3
4  </tr>
```

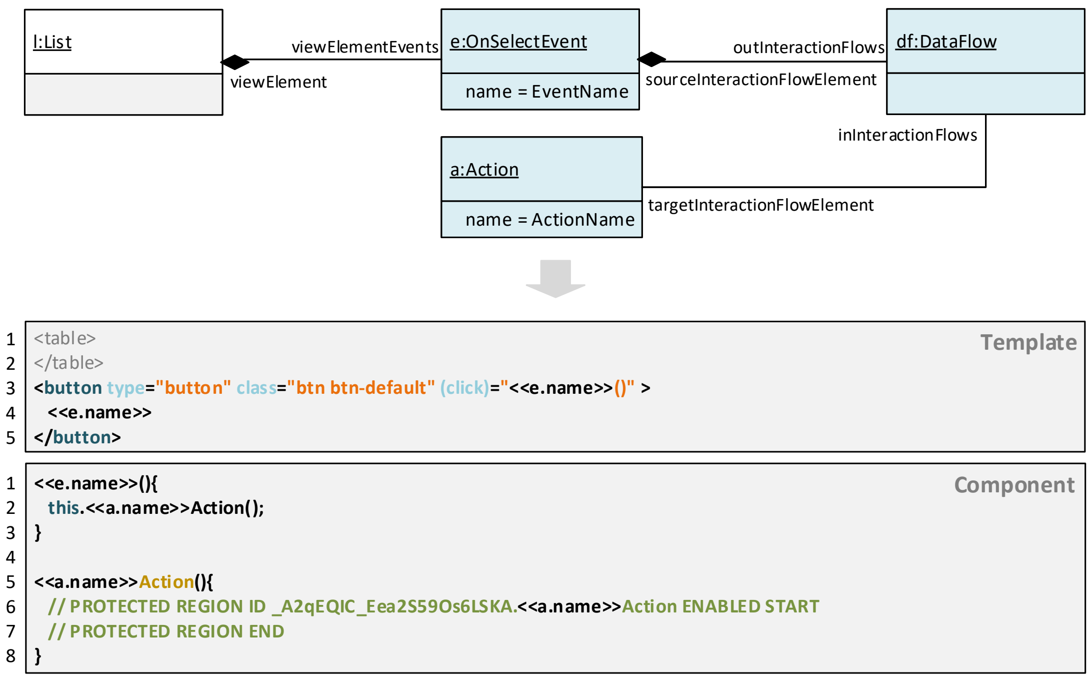
```
1  export class «vc.name»Component{
2    selected«p.name»: «p.type»;
3    isSelected«p.name»: boolean;
4
5    onSelect(el: DomainConcept){
6      if(this.selected«p.name» === el){
7        this.selected«p.name» = undefined;
8        this.isSelected«p.name» = false;
9      }else{
10       this.selected«p.name» = el;
11       this.isSelected«p.name» = true;
12     }
13   }
14 }
```

Fig. 5.6 Mapping for *Parameter* element

Description: *Parameters* are *InteractionFlowModelElements* and are part of an *InteractionFlowElement*. An *InteractionFlowElement* can be for example a *ViewElement* in form of a *List* (as shown in Figure 5.6). To establish a property binding for the parameter between the Angular *Component* and the *Template*, a variable declaration is needed in the Angular *Component*. The variable declaration creates a variable with name and type defined for the parameter in the *IFML* model (*Component* Line 2). Additionally, a flag is created that indicates whether the parameter is filled with information (*Component* Line 3). Within the *onSelect*-function, which is called when an element in the HTML table representing the list is selected, the variables for the parameter and the parameter flag are set (*Component* Line 6-11). In the *Template*, a conditional CSS class attribute is added for each row. The condition is satisfied if the data item presented in the row is the currently selected item, indicated by content of the parameter property binding (*Template* Line 2).

OnSelectEvents and *OnSubmitEvents* are mapped to elements in both, the Angular *Component* and the Angular *Template*. The mapping, however, is rather intuitive. In the *Template*, the events (meaning either *OnSelectEvent* or *OnSubmitEvent*) are created as HTML Button, calling a newly created function stub for the specific event in the Angular *Component*. There is no functionality added to those stubs as application code is not the main focus of this solution. The functionality comes with the associated interaction flow and its target element.

<u>Source Element:</u> *DataFlow* (from *OnSelectEvent* to *Action*)
Target Element: HTML Button, function for *OnSelectEvent* + stub for *Action*
Chronology: An *OnSelectEvent* is associated with a *List*. Within the generator for the *List*, a button for the event is added to the HTML *Template*. Functions for *Event* and *Action* are created during *ViewContainer* generation (see Figure 5.10).



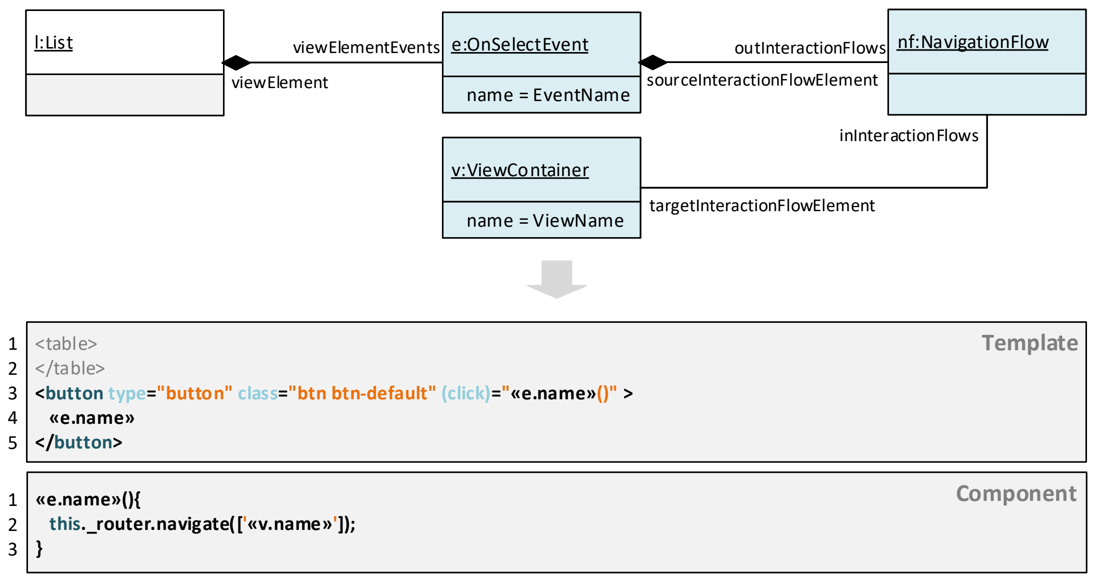
```
1 <table>
2 </table>
3 <button type="button" class="btn btn-default" (click)="<<e.name>>()" >
4   <<e.name>>
5 </button>
```

```
1 <<e.name>>(){
2   this.<<a.name>>Action();
3 }
4
5 <<a.name>>Action(){
6   // PROTECTED REGION ID _A2qEQIC_Eea2S59Os6LSKA.<<a.name>>Action ENABLED START
7   // PROTECTED REGION END
8 }
```

Fig. 5.7 *DataFlow* to *Action* mapping

Description: Figure 5.7 illustrates the mapping of a *DataFlow* from an *OnSelectEvent* to an *Action*. Within the *Component*, a function with the name of the *OnSelectEvent* is created (*Component* Line 1-3). The *DataFlow* is modeled by a call to a function stub created for the *Action*. The *Action* stub can be seen in *Component* Line 5-8. Since there is no *ParameterBindingGroup* associated with the *DataFlow*, there are no parameters transferred. Within the *Action* stub, a protected region is created. Protected regions allow the preservation of manually written code between code generation iterations. Within the *Template*, a button that calls the function stub for the event is created (*Template* Line 3-5).

Source Element: *NavigationFlow* (from *OnSelectEvent* to *ViewContainer*)
Target Element: HTML Button, function for *OnSelectEvent* with navigation to ViewContainer
Chronology: An *OnSelectEvent* is associated with a *List*. Within the generator for the *List*, a button for the event is added to the HTML *Template*. The function for the *Event* is created during *ViewContainer* generation.



```
1 <table>
2 </table>
3 <button type="button" class="btn btn-default" (click)="«e.name»()" >
4   «e.name»
5 </button>
```

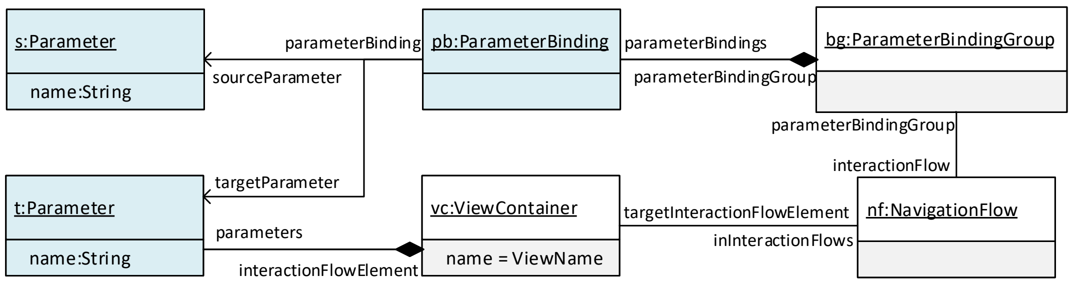
```
1 «e.name»(){
2   this._router.navigate(['«v.name»']);
3 }
```

Fig. 5.8 *NavigationFlow* to *ViewContainer* mapping

Description: In contrast to *DataFlows*, *NavigationFlows* (see Figure 5.8) can change the current view. The *targetInteractionFlowElement* of a *NavigationFlow* is a *ViewContainer*. The *OnSelectEvent* is mapped to a function with identical name in the Angular *Component*. The *NavigationFlow* is implemented in the *Component* as a call to the routing service of Angular. The routing service transfers the view focus to the target *ViewContainer*, however, no information is transferred between the views. To transfer information from one view to another, we need *ParameterBindings*. When a *ParameterBinding* is associated with the *NavigationFlow*, the source *Parameter* is added as URL parameter to the routing service navigation call. Again, a button is created within the *Template* to call the function stub in the *Component* (*Template* Line 3-5).

Source Element: *ParameterBinding* (on *NavigationFlow* for target *ViewContainer*)
Target Element: URL parameter retrieval

Chronology: Incoming *interactionFlows* are evaluated within the *ViewContainer* generation in the *ngOnInit-function*.



```
                                                                  Component
1   ngOnInit(){
2
3     // Incoming Navigation Flow with Parameter Binding
4     this._route.params.subscribe(params => {
5       if(params['«t.name»'] != undefined){
6         this.selectedBookToIssueBinding = JSON.parse(decodeURI(params['«t.name»']));
7       }
8     });
9
10    // PROTECTED REGION ID _tPI6IlaoEeaTJocisBH8lA.ngOnInit ENABLED START
11    // PROTECTED REGION END
12  }
```

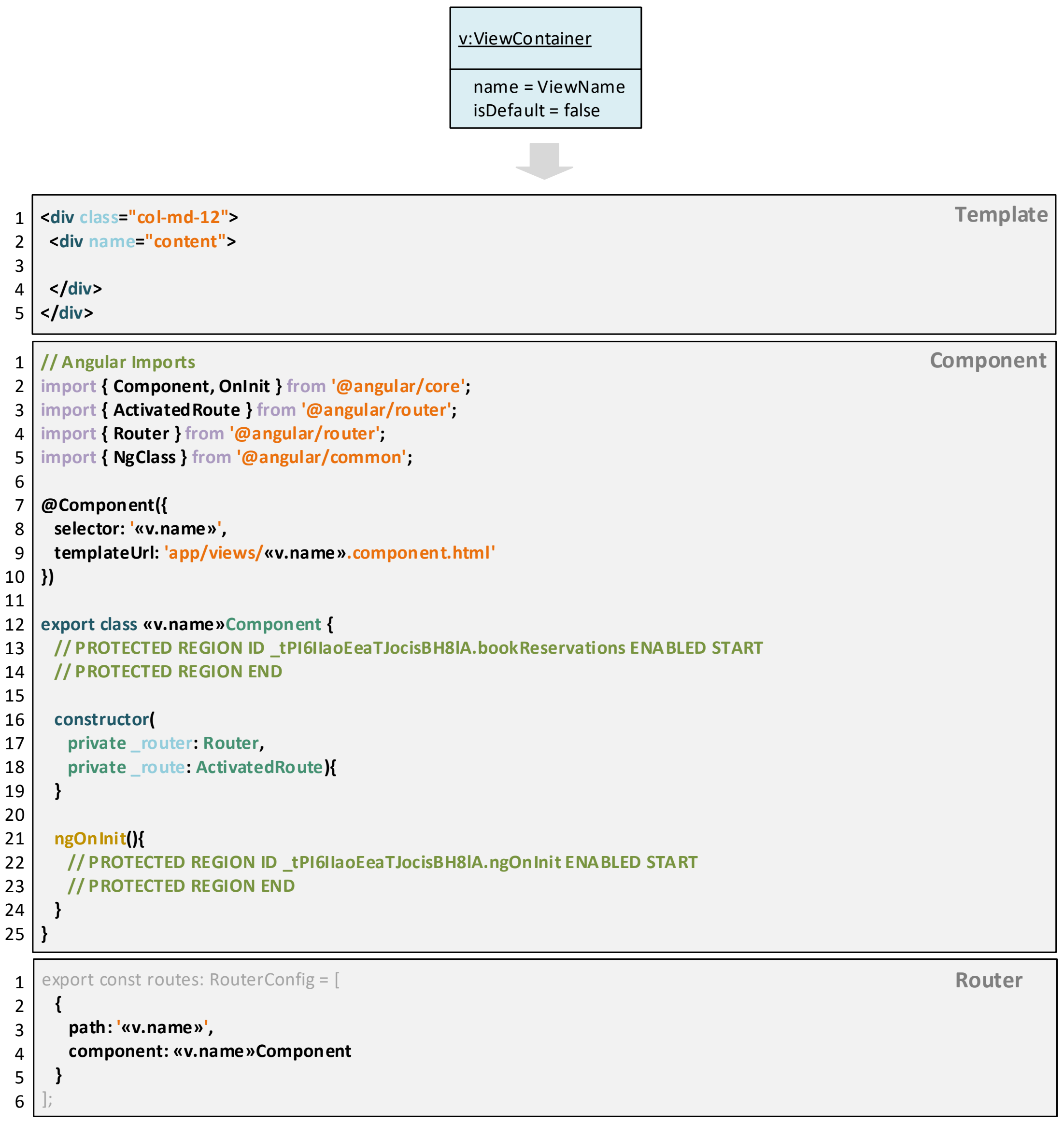
Fig. 5.9 Mapping for *ParameterBinding* element

Description: *NavigationFlows* in IFML can have associated *ParameterBindingGroups* (see Figure 5.9). This mapping will focus on an incoming *NavigationFlow* of a *ViewContainer* with *ParameterBinding*. The route service of Angular allows to subscribe to URL parameters that are transmitted to the view. This is done within the *ngOnInit*-function in the Angular *Component*. Each *ParameterBindingGroup* can have multiple *ParameterBindings* of type *Parameter*, however, no other child type is possible. For the sake of simplicity and a better overview, *ParameterBindingGroups* will therefore be considered as a simple element. The value of the *sourceParameter* is retrieved from the URL by the name of the *targetParameter* associated with the target *ViewContainer* (*Component* Line 4-8). It is then assigned to a variable within the Angular *Component*.

In the following, the IFML2Angular mappings are described for the more complex *IFML* model elements such as *ViewContainer*, *ViewComponent*, and *DataBinding*.

Complex elements

In contrast to the previously explained simple element mappings from IFML to Angular, the elements in this section are more complex, as they can be the parent container of other *IFML* model elements. We start with the description of an *IFML2Angular* mapping for the most relevant *IFML* model element *ViewContainer*.



```html
<div class="col-md-12">
  <div name="content">

  </div>
</div>
```

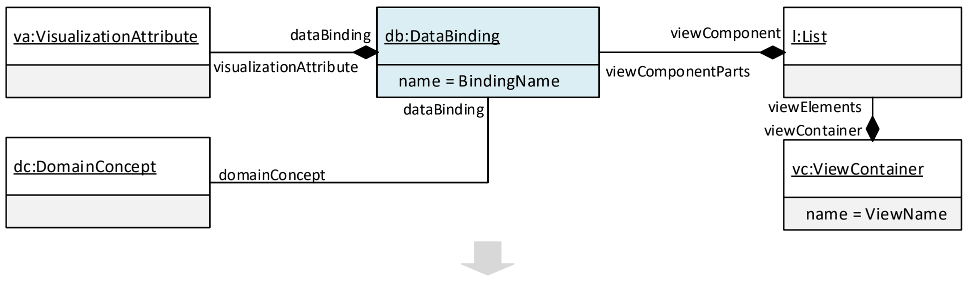
```typescript
// Angular Imports
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Router } from '@angular/router';
import { NgClass } from '@angular/common';

@Component({
  selector: '«v.name»',
  templateUrl: 'app/views/«v.name».component.html'
})

export class «v.name»Component {
  // PROTECTED REGION ID _tPI6IlaoEeaTJocisBH8lA.bookReservations ENABLED START
  // PROTECTED REGION END

  constructor(
    private _router: Router,
    private _route: ActivatedRoute){
  }

  ngOnInit(){
    // PROTECTED REGION ID _tPI6IlaoEeaTJocisBH8lA.ngOnInit ENABLED START
    // PROTECTED REGION END
  }
}
```

```typescript
export const routes: RouterConfig = [
  {
    path: '«v.name»',
    component: «v.name»Component
  }
];
```

Fig. 5.10 Mapping for *ViewContainer* element

<u>**Source Element:**</u> *ViewContainer*
Target Element: Angular *Component* and *Template* base structure, *Angular Routes*
Chronology: Chronologically, the *ViewContainer* is the first executed transformation in the UI Generator as all child elements of the *ViewContainer* are called from within the *ViewContainer* generator (see Section 5.2.3 for further implementation details).
Description: The *ViewContainer* (see Figure 5.10) is the parent container of most *IFML* elements used in this thesis. On the Angular side, the *IFML ViewContainer* is equivalent to a view, meaning the Angular *Component* as well as the *Template*. Additionally, the set of *ViewContainers* determines the routing within the Angular application. The *Template* created for a *ViewContainer* is just the base structure of the HTML representation with two nested div-elements, which only carry some layout information (*Template* Line 1-5). The Angular *Component* carries some more information. Firstly, the imports of required classes and modules are defined (*Component* 2-5). Then, there is the *@Component* declaration, which defines the selector of the *Component* and the location of the *Template* (*Component* Line 7-10). Both are based on the name of the *ViewContainer*. The class declaration, created again with the name of the *ViewContainer*, contains a protected region, a constructor and an Angular initialization method (*ngOnInit*). The protected region is used to allow developers implementing manually written code which is preserved, even on new generations. The constructor receives navigation centered Angular services as input (*Component* Line 16-19). The initialization method (*Component* Line 21-24), initially, contains only another protected region to make manual adjustments to the view on initialization. There are other parts of code generated within the *ViewContainer* generator, but they depend on other, specific *IFML* model elements, and as such are explained in the mapping of those elements. Of special consideration is also the creation of *Routes*. The navigation between views is managed by URLs, which are mapped to a single view, represented by a *Component* (Router Line 3-4). Path and, as previously explained, *Component* name are derived from the *ViewContainer* name. A special case, not depicted in the shown mapping, is the *DefaultViewContainer*. In the *RouterConfig* a default path can be set, if the URL does not contain a path to a view. The default path then redirects to the path of the (unique) *DefaultViewContainer*.

<u>**Source Element:**</u> *DataBinding*
Target Element: HTML Table content binding, Function stub and property binding for data retrieval in *Component*
Chronology: The HTML Table content binding is done within the *ViewComponent* generation. Function stub and property binding are created within *ViewContainer* generation.



Template
```
1 <tr *ngFor="#el of <<db.name>>| exampleListFilter: filterBy" (click)="onSelect(el)"
2    [class.info]="el === selectedParameterBinding">
3
4 </tr>
```

Component
```
1  export class <<vc.name>>Component{
2    // variable for list data binding
3   <<db.name>>: <<db.domainConcept>>[];
4
5    // stubs for data service calls for data bindings
6    get<<db.name>>(){
7      // PROTECTED REGION ID _zM55ElaoEeaTJocisBH8lA.get<<db.name>> ENABLED START
8      // PROTECTED REGION END
9    }
10
11   ngOnInit(){
12     // Call stub for filling data binding
13     this.get<<db.name>>();
14   }
15 }
```

Fig. 5.11 Mapping for *DataBinding* element

Description: *DataBindings* in Lists and Details in *ViewComponents* basically have the same functional behavior. Therefore, only *DataBindings* associated with *Lists* are explained in detail. *DataBindings* are specifying the data which should be displayed in the *ViewComponent* (see Figure 5.11). The type of data is defined by the associated *DomainConcept*. When it gets to mapping, the property binding between *Template* and *Component* is important. In the *Template*, the property binding is done in the for-loop (ngFor) for creating the different table rows (*Template* Line 1). The variable referred to in the for-loop is also created in the *Component* as an array of objects of the specified *DomainConcept* (*Component* Line 3). The array is supposed to be filled in the function stub containing a protected region (*Component* Line 6-9). Within this protected region, the developer can make calls to the service managing the data access. The stub is called automatically as soon as the view is initialized (*Component* Line 13). The *VisualizationAttributes* are dependent on the type of the *View-Component*, and as such, are created within the *ViewComponent* generation. They specify, which attributes of the specified *DomainConcept* of the *DataBinding* are presented to the user.

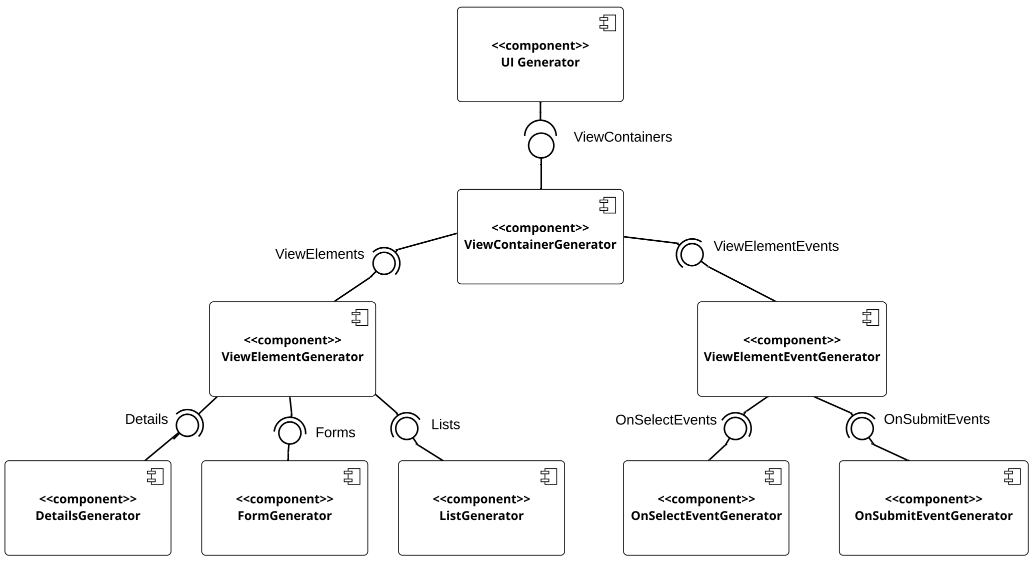
Source Element: *List*
Target Element: HTML Table
Chronology: The generator to generate the HTML Table is called, when the *ViewContainer* template is created. The imports to support the search and filtering of the generated table are created within the *ViewContainer* generator.
Description: The *ListViewComponent* is represented as an HTML Table in the Angular *Template*. The displayed data is retrieved from the variable created for the *DataBinding*. Columns in the table, however, are only created for the *VisualizationAttributes* associated with the *DataBinding*. The generation of the *OnSelectEvents* creates HTML buttons in the *Template*, as well as the corresponding function stubs in the *Component*. The HTML table also provides functionality, which allows filtering and searching the elements of the *DataBinding*. For this, some requirements need to be imported into the Angular *Component* in form of directives and pipes. The search directive is also added to the Angular *Template*. Another feature of the HTML table is the ability to switch between the table view and a special details view for mobile devices (*Mobile Details View*). This special mobile view element is also added to the HTML *Template*.

Based on the above described mappings, we have implemented a *UI Generator* which takes as input the domain and IFML model and creates the *Final UI* (FUI) code in an automated way. At this point, it should be noticed that our generation approach was illustrated based on a subset of IFML model elements. Although it supports the generation of quite representative FUIs based on Angular, its main objective is not to be complete in the sense that all *IFML* and *Angular* target elements are fully covered. Beside that, it is important to mention that the defined mappings are focusing on Angular as target Web/UI framework. However, due to the flexibility offered by template-based code generators (see Section 2.6.3), it is easily possible to define and establish a mapping from IFML to further target technologies to enable the generation of *Final UI* code for other UI frameworks. With this regard, *IFML* provides a mapping specification[4] to other frameworks such as .NET's Windows Presentation Foundation (WPF) or Java Swing. In the following, we elaborate on the implementation of the *UI Generator* and describe how the defined mappings were used to implement a template-based code generator for *FUI*s in Agular.

[4]http://www.ifml.org/wp-content/uploads/IFML-PSM-mappings.pdf

5.2.3 Implementation of *UI Generator*

The technical implementation of the *UI Generator* is realized with Xtend[5] and is based on the previously defined mappings. As introduced in Section 2.6.3, Xtend supports template expressions to enable the easy implementation of a template-based code generation approach. In a template-based generation, predefined templates are filled with information from the model to generate the target code. In our case, this means that for every *IFML* element, depending on its type, one or more Xtend templates, either for the Angular *Template*, the Angular *Component* or both, have to be defined and filled with information extracted from the *IFML* model element. There are different supporting *Classes* to realize such a template-based code generation approach based on Xtend. *ClassGenerators* allow the creation of the Angular *Component* and *Template* file by utilizing one method for the creation of each Angular part. *CodeGenerators* contain a function to just create an Xtend template as return value, without creating a file. *FileGenerators* can create a file of arbitrary type and content. Figure 5.12 shows the general structure of the *UI Generator*. As it can be seen, the *UI Generator* calls for each *IFML ViewContainer* in the *IFML* model, the *ViewContainerGenerator*. The *ViewContainerGenerator* is responsible for creating the basic structure of both the Angular *Component* and *Template*.



<<component>> UI Generator
ViewContainers
<<component>> ViewContainerGenerator
ViewElements
ViewElementEvents
<<component>> ViewElementGenerator
<<component>> ViewElementEventGenerator
Details
Forms
Lists
OnSelectEvents
OnSubmitEvents
<<component>> DetailsGenerator
<<component>> FormGenerator
<<component>> ListGenerator
<<component>> OnSelectEventGenerator
<<component>> OnSubmitEventGenerator

Fig. 5.12 Architectural overview of the *UI Generator*

The information that is filled into the Angular *Template* is either directly extracted from the *IFML ViewContainer* element or created by calling another generator for the

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

specific element. As previously described, the mapping used for the transformation of the *ViewContainer* element can be seen in Figure 5.10.

Each subcomponent of the *UI Generator* is implemented through an Xtend template characterizing the before mentioned IFML model to Angular element mappings on a technical implementation level. As most of the Xtend templates are quite exhausting and require lots of space, example templates are presented in the following to give a better impression about the implementation of the UI generation approach.

Code Excerpt 5.1 shows a part of the *ViewContainerGenerator* Xtend template (see also the corresponding mapping in Figure 5.10). The *ViewContainerGenerator* creates at first the necessary Angular core imports (Line 3-6). These are static and include, for example, the Angular services responsible for routing or the component definition. After that, depending on the contained *IFML ViewComponents*, further import statements, which are required by the *IFML ViewComponents*, are created. If there is a *ViewComponent* of type *List* for example (Line 9), certain components to enable the filtering of the items within the *List* are needed (see mapping for list element).

```
 1  class ViewContainerGenerator extends AbstractClassGenerator<
       ViewContainerImpl> {
 2  // Angular Imports
 3  import { Component, OnInit } from ’@angular/core’;
 4  import { ActivatedRoute } from ’@angular/router’;
 5  import { Router } from ’@angular/router’;
 6  import { NgClass } from ’@angular/common’;
 7
 8  «FOR vElem : it.viewElements»
 9      «IF (vElem instanceof ListImpl)»
10          // Search Component Imports
11          import { SearchComponent } from ’../dynamic/search.component
               ’;
12          import { «vElem.name.toFirstUpper»Filter } from ’../helper/
               pipes/«vElem.name.toFirstLower».pipe’;
13      «ENDIF»
14  «ENDFOR»
15  // Service Imports
16  «FOR service : ServiceCollection.sharedInstance.services»
17      import { «service.name» } from ’..«service.location»’;
18  «ENDFOR»
19  ...
20  }
```

Code Excerpt 5.1 Xtend template excerpt from *ViewContainerGenerator*

First of all, there is the *SearchComponent* (Line 11), which is a directive that provides the input fields for the filter value. Additionally a filter, or pipe, is imported (Line 12). The filter is generated specifically for each *ViewComponent* of type *List*.

The generation of the different filters is orchestrated from within the *CodeGenerator* class. Also, the import statements for different services (Line 16-18), which are statically defined, are created. These services include general functions like user authentication and which are then used within the transformation templates.

A further Xtend template example is shown in Code Excerpt 5.2. The *ViewElementGenerator* takes all the *IFML* View Components as a list of parameter. Within the *ViewElementGenerator*, for each *IFML View Component* it is decided, which specialized generator is called. We have implemented specific generators for each *IFML View Component* such as *Details*, *Forms*, and *Lists*.

```
public class ViewElementGenerator {

    protected def String generateCode(List<ViewElement>
        viewElementList){
        var output = ""

        for (viewElement : viewElementList){
            switch viewElement {
                ListImpl: output += new ListGenerator().
                    generateTemplate(viewElement)
                FormImpl: output += new FormGenerator().
                    generateTemplate(viewElement)
                DetailsImpl: output += new DetailsGenerator().
                    generateTemplate(viewElement)
                default: output += ""
            }
        }
        return output
    }
}
```

Code Excerpt 5.2 Xtend template excerpt from *ViewElementGenerator*

The *DetailsGenerator*, *FormGenerator*, and *ListGenerator* are creating code as output without creating a new file. Since it is mostly generated HTML code, no code excerpt is listed here. The DetailsGenerator creates an HTML table for the *IFML Details* with fields created with regards to the *IFML Data Binding*. The *IFML Visualisation Attributes* of the *IFML Data Binding* determine which attributes are created as fields in the table. The *FormGenerator* works in a similar way and creates also a table for displaying the *Visualisation Attributes* in the *IFML List*. However, the table has a traditional layout, opposed to the tilted layout in the *DetailsGenerator*, where the table headers are the first column and the elements are the second column.

Furthermore, HTML buttons for *IFML View Element Events* are added. For this purpose, the *ViewElementEventGenerator* calls the specialized generators *OnSelectEventGenerator* and *OnSubmitGenerator* (see Code Excerpt 5.3). Both generators, *OnSelectEventGenerator* and *OnSubmitEventGenerator* create function stubs for the events in the Angular *Component* with protected regions.

```
public class ViewElementEventGenerator {
    protected def String generateCode(List<ViewElement>
       viewElementList){
        var output = ""
        for (viewElement : viewElementList){
            switch viewElement {

                // OnSelect
                ListImpl: output += new OnSelectEventGenerator().
                    generateCode(viewElement)
                // OnSubmitEvent
                FormImpl: output += new OnSubmitEventGenerator().
                    generateCode(viewElement)

                default: output += ""
            }
        }
        return output
    }
}
```

Code Excerpt 5.3 Xtend template excerpt from *ViewElementEventGenerator*

To sum up, based on the above described transformation approach using a template-based code generation technique based on Xtend, we were able to automatically generate the needed UI views for our model-driven engineering approach for self-adaptive UIs. As the UI on its own provides no context-management and UI adaptation capabilities, in the following subsections we present how to automatically generate *Context* and *Adaptation Services*.

5.3 Context Service Generation

To address the requirement *R2 'Context' Transformation Approach* introduced in Section 3.2, we will describe the generation approach for a *Context Service*. The goal of the *Context Service Generator* is the automated creation of an Angular service that allows context monitoring of specified context-of-use parameters at runtime using hardware sensors of the target platform.

In the following, we first describe the mappings between *ContextML* and Angular services for supporting the transformation process. After that, we describe implementation related details about the *Context Service* generation.

5.3.1 Mapping: *ContextML2AngularServices*

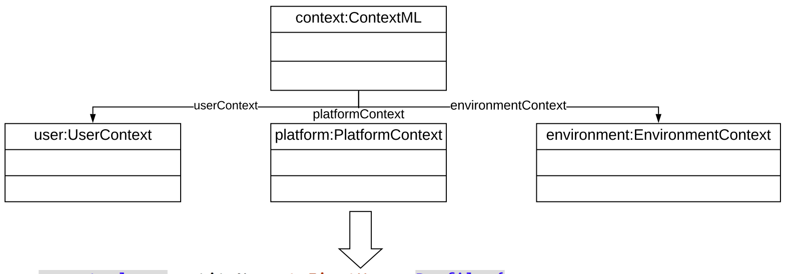
The *Context Service Generator* gets as input the previously mentioned *Context Model* which is based on *ContextML*. The following Table 5.2 depicts on the left column the main source elements of *ContextML* that has to be translated to derive an *Injectable ContextService*.

Source Element	**Target Element**
ContextML	*Injectable ContextService*
ContextEntity	*ContextProfile*
ContextProvider	*ContextProviderService*
ContextTypeEnumeration	*ContextEnum*

Table 5.2 *ContextML2AngularService* Mappings for *Context Service Generation*

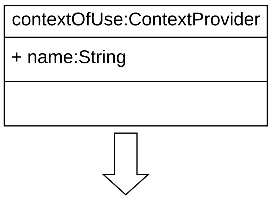
The core elements of *ContextML* to specify context-of-use parameters are *ContextEntity* which is transformed to a *ContextProfile*, *ContextProvider* which is transformed to a *ContextProviderService*, and finally *ContextTypeEnumeration* which is transformed to a *ContextEnum*. For each of the source *ContextML* model element we have defined a translation mapping in to the target technology characterizing the *Context Service* in Angular.

As depicted in Figure 5.13, the context-of-use parameter information from each context entity *UserContext*, *PlatformContext*, and *EnvironmentContext* is accessed and stored in a generic context profile where the context data are persisted conform to the specified data types in the conext model. Furthermore, Figure 5.14 shows, how *ContextProvider* hardware sensors are requested to receive the context-of-use data that were specified before. The last mapping from *ContextTypeEnumeration* to *ContextEnum* is not illustrated here as it is characterized by a trivial mapping based on enumeration classes.



```
export class «entityName.toFirstUpper»Profile {
«FOR prop: properties»
    «var propName = prop.name»
    «var propType =  prop.eClass.EAllAttributes
    .findFirst[attr|attr.name == 'value']?.EType?.name.toFirstUpper»
        private «propName»: «
            if(propType == "EString"){propType = "string";}
            else if(propType == "EInt"){propType = "number";}
            else if(propType == "EString"){propType = "string";}
            else if(propType == "Double"){propType = "number";}
            else if(propType == "EDouble"){propType = "number";}
            else if(propType == "EBoolean"){propType = "boolean";}
            else{propType}»;
«ENDFOR»
    constructor(){};
«FOR prop: properties»
}
```

Fig. 5.13 *ContextEntity2ContextProfile* Mapping

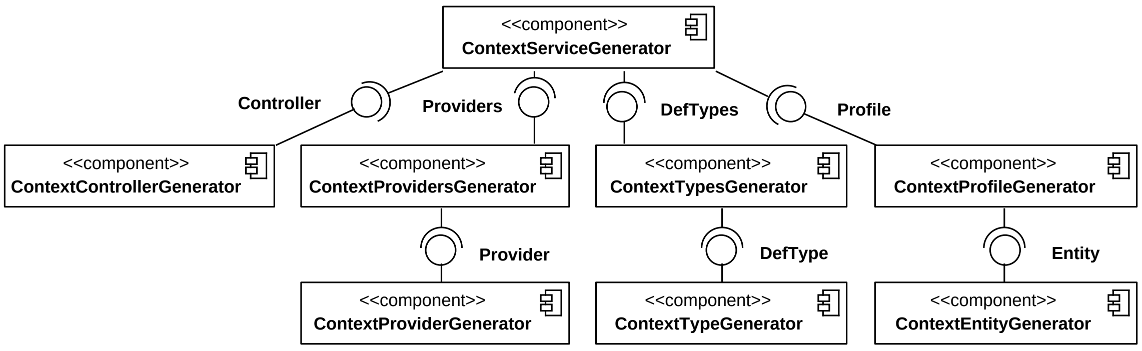


```
export class «providerName.toFirstUpper»Service {

    Subjects for the properties are created
    «FOR prop: propertyList»
    « val moodEnabled = propertyList
    .filter[prop|prop.eClass.name == "Mood"].length>0»
    « val genderEnabled = propertyList
    .filter[prop|prop.eClass.name == "Gender"].length>0»
    « val AgeCalculatedEnabled = propertyList
    .filter[prop|prop.eClass.name == "AgeCalculated"].length>0»
    «IF genderEnabled || moodEnabled || AgeCalculatedEnabled»

    // PROTECTED REGION ID «providerName.toFirstLower» ENABLED START
    // PROTECTED REGION END
    «IF propertyList.filter[prop|prop.eClass.name == "Activity"].length>0»
    «IF propertyList.filter[prop|prop.eClass.name == "Weather"].length>0»
    constructor(){
        const GenericProviderService=this;
        «IF propertyList.filter[prop|prop.eClass.name == "NoiseLevel"].length>0»
        «IF propertyList.filter[prop|prop.eClass.name == "Language"].length>0»
        «IF propertyList.filter[prop|prop.eClass.name == "Time"].length>0»
        «IF propertyList.filter[prop|prop.eClass.name == "Timezone"].length>0»
        «IF propertyList.filter[prop|prop.eClass.name == "Date"].length>0»
        «IF propertyList.filter[prop|prop.eClass.name == "Activity"].length>0»
        «IF genderEnabled || moodEnabled || AgeCalculatedEnabled»
        «IF propertyList.filter[prop|prop.eClass.name == "Light"].length>0»
        «IF propertyList.filter[prop|prop.eClass.name == "UsageTime"].length>0»
        Here is the place where code for EventListeners
    }
    Add callback functions for the properties
}
```

Fig. 5.14 *ContextProvider2ContextProviderService* Mapping

5.3.2 Implementation of *Context Service Generator*

The structure of the *ContextServiceGenerator* is shown in Figure 5.15. It has a main generator that splits the generation into four kinds of files that will be generated: *ContextControllerGenerator*, *ContextProvidersGenerator*, *ContextTypesGenerator*, and *ContextProfileGenerator*. Our *ContextServiceGenerator* is a template-based code generator that is also implemented based on *Xtend*[6].



<<component>> ContextServiceGenerator
Controller
Providers
DefTypes
Profile
<<component>> ContextControllerGenerator
<<component>> ContextProvidersGenerator
<<component>> ContextTypesGenerator
<<component>> ContextProfileGenerator
Provider
DefType
Entity
<<component>> ContextProviderGenerator
<<component>> ContextTypeGenerator
<<component>> ContextEntityGenerator

Fig. 5.15 Architectural overview of the *Context Service Generator*

Firstly, as depicted in Code Excerpt 5.4, the *ContextServiceGenerator* invokes the *ContextControllerGenerator*, which generates the main Angular service that connects and controls all the other parts. The generated *ContextController* contains subscriptions to context properties, which push changed data automatically to the subscriber based on the *RxJS observer pattern*[7]. Furthermore, it contains timers for the properties which are not updated in an event-based manner.

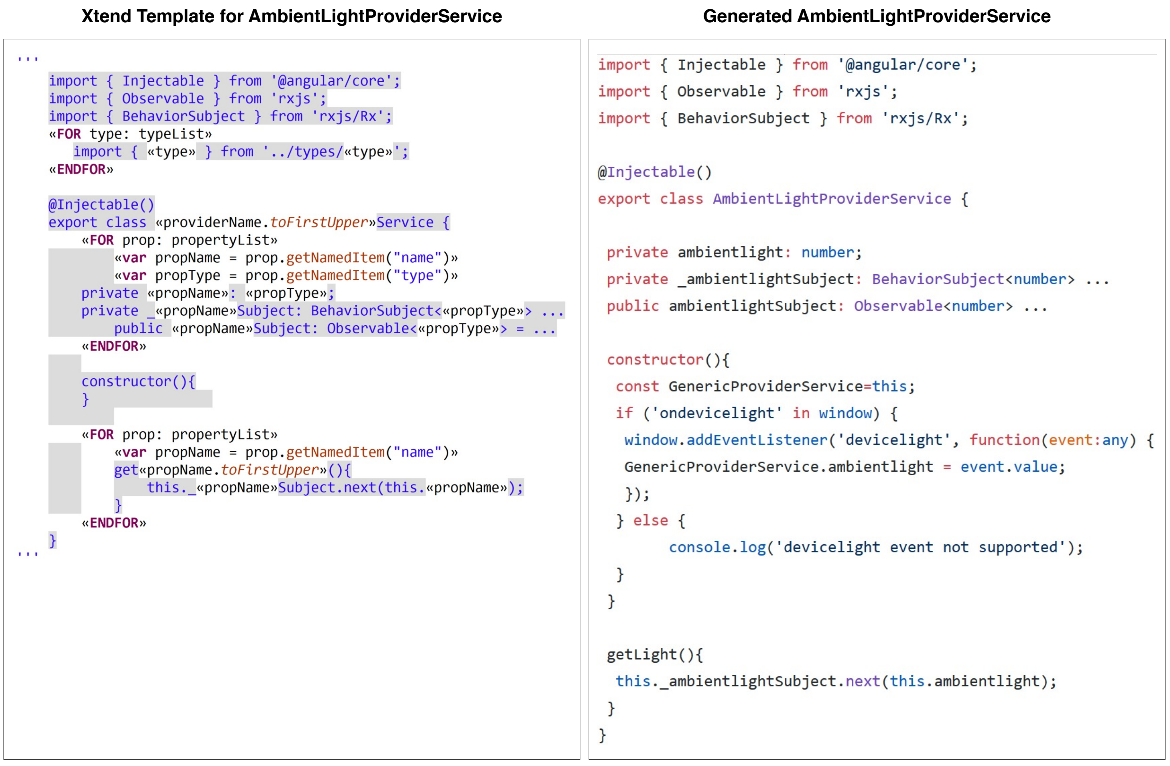
```
class ContextServiceGenerator{
    def generateFiles(ContextML context){

        // Invoke the four parts of the Context Service Generator
        new ContextControllerGenerator().generateFile(context);
        new ContextProvidersGenerator().generateFiles(context);
        new ContextTypesGenerator().generateFiles(context);
        new ContextProfileGenerator().generateFile(context);
    }
}
```

Code Excerpt 5.4 Xtend template excerpt from *ContextServiceGenerator*

[6]http://www.eclipse.org/xtend/
[7]https://github.com/Reactive-Extensions/RxJS

The *ContextProvidersGenerator* invokes the *ContextProviderGenerator* for each provider that is listed in the *Context Model*. This creates a folder with all provider files. Each file contains standard imports and used *DefTypes*. The business logic code for controlling and managing of sensor sources, like APIs or libraries has to be inserted manually. This is due to the very individual structure of numerous interfaces. Those can be fairly easy to use, like standard HTML5 APIs[8], but can be individual and more complex as well, like the Affectiva SDK for emotion recognition.
Similarly, the *ContextTypesGenerator* invokes the *ContextTypeGenerator* for each user defined *DefType*. This creates a folder with type files, that are imported by the providers. Each file contains the *Enums* defined in the *Context Model*.

The last generator component is the *ContextProfileGenerator* that creates a central context data profile file and invokes the *ContextEntityGenerator* for each declared entity in the *Context Model*. This creates a file for each entity which contains all the defined properties and the corresponding *getter* and *setter* methods. The generated *Context Service* files are injected into the Angular UI framework as modular components.



Xtend Template for AmbientLightProviderService

```
'''
    import { Injectable } from '@angular/core';
    import { Observable } from 'rxjs';
    import { BehaviorSubject } from 'rxjs/Rx';
    «FOR type: typeList»
       import { «type» } from '../types/«type»';
    «ENDFOR»

    @Injectable()
    export class «providerName.toFirstUpper»Service {
        «FOR prop: propertyList»
            «var propName = prop.getNamedItem("name")»
            «var propType = prop.getNamedItem("type")»
        private «propName»: «propType»;
        private _«propName»Subject: BehaviorSubject<«propType»> ...
            public «propName»Subject: Observable<«propType»> = ...
        «ENDFOR»

        constructor(){
        }

        «FOR prop: propertyList»
            «var propName = prop.getNamedItem("name")»
            get«propName.toFirstUpper»(){
                this._«propName»Subject.next(this.«propName»);
            }
        «ENDFOR»
    }
'''
```

Generated AmbientLightProviderService

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { BehaviorSubject } from 'rxjs/Rx';

@Injectable()
export class AmbientLightProviderService {

 private ambientlight: number;
 private _ambientlightSubject: BehaviorSubject<number> ...
 public ambientlightSubject: Observable<number> ...

 constructor(){
  const GenericProviderService=this;
  if ('ondevicelight' in window) {
   window.addEventListener('devicelight', function(event:any) {
   GenericProviderService.ambientlight = event.value;
   });
  } else {
        console.log('devicelight event not supported');
  }
 }

 getLight(){
  this._ambientlightSubject.next(this.ambientlight);
 }
}
```

Fig. 5.16 Xtend template excerpt for ContextProviderGenerator and its generated code

[8]https://www.w3.org/2009/dap/

An example for the generation of a *Context Provider* is depicted in Figure 5.16. The code excerpt depicted in Figure 5.16 represents on the left side the Xtend template for generating a specific context provider for capturing the ambient light level through a sensor library. On the right side of Figure 5.16, the generated code for the *AmbientLightProviderService* is illustrated. The code of the generated context provider is responsible for monitoring the environmental lighting condition at runtime by using the *AmbientLightAPI*.

5.4 Adaptation Service Generation

For addressing the requirement *R6 'Adaptation' Transformation Approach* introduced in Section 3.2, we describe the generation approach for an *Adaptation Service*. The goal of the *Adaptation Service Generator* is the automated creation of an Angular service that allows UI adaptation at runtime. UI adaptations, as introduced in the context of our modeling framework (see Chapter 4), are expressed in a rule-based manner using *AdaptML*. Based on this input file, the *Adaptation Service Generator* generates an Angular service containing the JavaScript rule engine Nools[9]. Nools is an efficient RETE-based rule engine written in JavaScript and provides an API for specifying facts and rules (see Section 2.6.2 for further details). In the following, analogously to the previous section, first the mappings between *AdaptML* and Angular services are described. After that, we describe implementation related details about the *Adaptation Service* generation.

5.4.1 Mapping: *AdaptML2AngularServices*

Firstly, we give an overview to characterize the automatic translation from abstract UI adaptation rules represented in *AdaptML* to a *Nools* service that represent the UI adaptation logic at runtime. The following Table 5.3 depicts on the left column the main source elements of *AdaptML* that has to be translated to derive an *Injectable NoolService*.

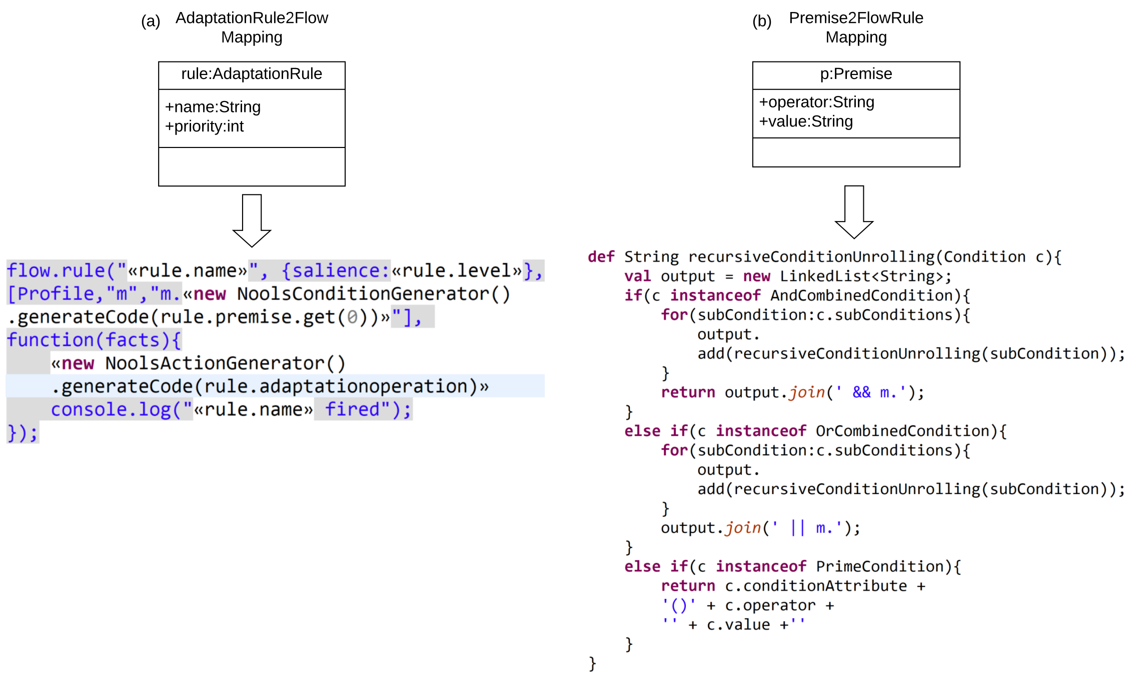
Source Element	**Target Element**
AdaptML	*Injectable NoolsService*
AdaptationRule	*Flow*
Premise (Condition)	*FlowRule*
AdaptationOperation	*NoolsAction*

Table 5.3 *AdaptML2AngularService* Mappings for *Adaptation Service Generation*

The core elements of *AdaptML* to specify UI adaptations are *AdaptationRule* which is transformed to a *Flow*, *Premise (Condition)* expressions which are transformed to a *FlowRule*,

[9] http://noolsjs.com/

and finally *AdaptationOperation* which is transformed to a *NoolsAction*. For each of the source *AdaptML* model element we have defined a translation mapping into the target technology of the used rule engine *Nools*. Figure 5.17 (left) depicts the mapping of an *AdaptationRule* to a *Flow*. In this case, the mapping is quite straight forward as each specified *AdaptationRule* is inserted into a *Nools* session as a *Flow*. Each *Flow* has a name and a salience which conforms to the *AdaptationRule's* attributes *name* and *priority*, respectively. The *Premise (Condition)* part of a UI adaptation rule is translated to a *FlowRule*. As shown on the right side of Figure 5.17, the mapping of the *AdaptML* conditions into *Nools* conditions is processed based on the *PrimeCondition* and structure of the *CombinedCondition*. Similarly, each *AdaptationOperation* is inserted as a *NoolsAction* into an *actionNodeList* which is then provided to *Nools*.



```
flow.rule("«rule.name»", {salience:«rule.level»},
[Profile,"m","m.«new NoolsConditionGenerator()
.generateCode(rule.premise.get(0))»"],
function(facts){
    «new NoolsActionGenerator()
    .generateCode(rule.adaptationoperation)»
    console.log("«rule.name» fired");
});
```

```
def String recursiveConditionUnrolling(Condition c){
    val output = new LinkedList<String>;
    if(c instanceof AndCombinedCondition){
        for(subCondition:c.subConditions){
            output.
            add(recursiveConditionUnrolling(subCondition));
        }
        return output.join(' && m.');
    }
    else if(c instanceof OrCombinedCondition){
        for(subCondition:c.subConditions){
            output.
            add(recursiveConditionUnrolling(subCondition));
        }
        output.join(' || m.');
    }
    else if(c instanceof PrimeCondition){
        return c.conditionAttribute +
        '()' + c.operator +
        '' + c.value +''
    }
}
```

Fig. 5.17 Example Mappings: (a) *AdaptationRule2Flow* and (b) *Premise2FlowRule*

5.4.2 Implementation of *Adaptation Service Generator*

The *Adaptation Service Generator*, which is synonymously called *NoolsServiceGenerator* in this work, due to the name of the used rule engine, is implemented with Xtend and takes the UI adaptation rules as input. Structurally, as shown in Figure 5.18, it consists of the components *NoolsServiceGenerator*, *NoolsRuleGenerator*, *NoolsConditionGenerator*, and

NoolsActionGenerator. These components are responsible for creating an injectable Angular service conform to the above described mappings.

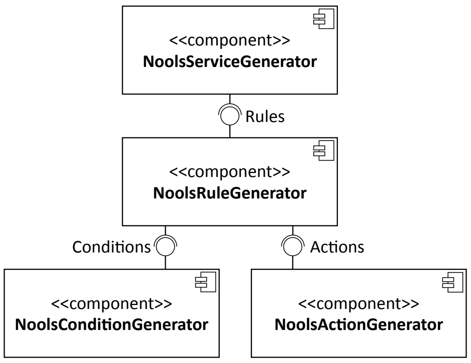


Fig. 5.18 *Adaptation Service Generation*

In the following, each component involved in the *Adaptation Service* generation is described based on its realized Xtend template.

Code Excerpt 5.5 shows a part of the Xtend template for the *NoolsServiceGenerator*.

```
class NoolsServiceGenerator extends AbstractFileGenerator<AdaptML> {
  ...
    @Injectable()
            export class NoolsService {
                private flow;
                private m: Profile;
                constructor(
                    private dcl: DynamicComponentLoader,
                    private injector: Injector,
                    private _Router: Router,
                    private _LoggerService: LoggerService,
                    private _ResourceService: ResourceService){
                    this.flow = nools.flow("«flow.attributes.
                        getNamedItem("name").nodeValue»", function(
                        flow){«new NoolsRuleGenerator().generateCode(
                        flow.childNodes, serviceMap, functionMap)»});
                }
    ...
}
```

Code Excerpt 5.5 Xtend template excerpt from *NoolsServiceGenerator*

It creates an injectable *Nools* service which consists of the required Angular imports, the class declaration of the service and the implementation of the *Nools* flow. The flow is composed of all the rules defined in the abstract UI adaptation rules based on *AdaptML*. For each rule it is defined under which conditions the rule actions are executed. The generation of the individual rules is delegated to the *NoolsRuleGenerator*.

The *NoolsRuleGenerator's* Xtend template is shown in Code Excerpt 5.6. For each adaptation rule it creates a *Nools* flow where the name of the *Adaptation Service* is the name of the abstract UI adaptation rule. The salience of the rule is the priority level of the rule and corresponds to the level defined in the *AdaptML* rule specification. In addition to that, the rule fact is defined by the *factType* and *factName* attributes. The generation of the conditions and adaptation operations of the rule is delegated to the *NoolsConditionGenerator*and the *NoolsActionGenerator*, respectively.

```
public class NoolsRuleGenerator{
    ...
    def protected generateTemplate(AdaptationRule[] ruleset) {
        var output = ""
        for(rule: ruleset){
                output += '''
                    flow.rule("«attr.getNamedItem("name").nodeValue»
                       ", {salience:«attr.getNamedItem("priority").
                       nodeValue»},[«attr.getNamedItem("factType").
                       nodeValue»,"«attr.getNamedItem("factName").
                       nodeValue»","«new NoolsConditionGenerator().
                       generateCode(rule.firstChild)»"], function(
                       facts){
                «new NoolsActionGenerator().generateCode(rule.
                    firstChild.nextSibling.childNodes, serviceMap,
                    functionMap)»});
                '''
        }
        return output
    }
    ...
}
```

Code Excerpt 5.6 Xtend template excerpt from *NoolsRuleGenerator*

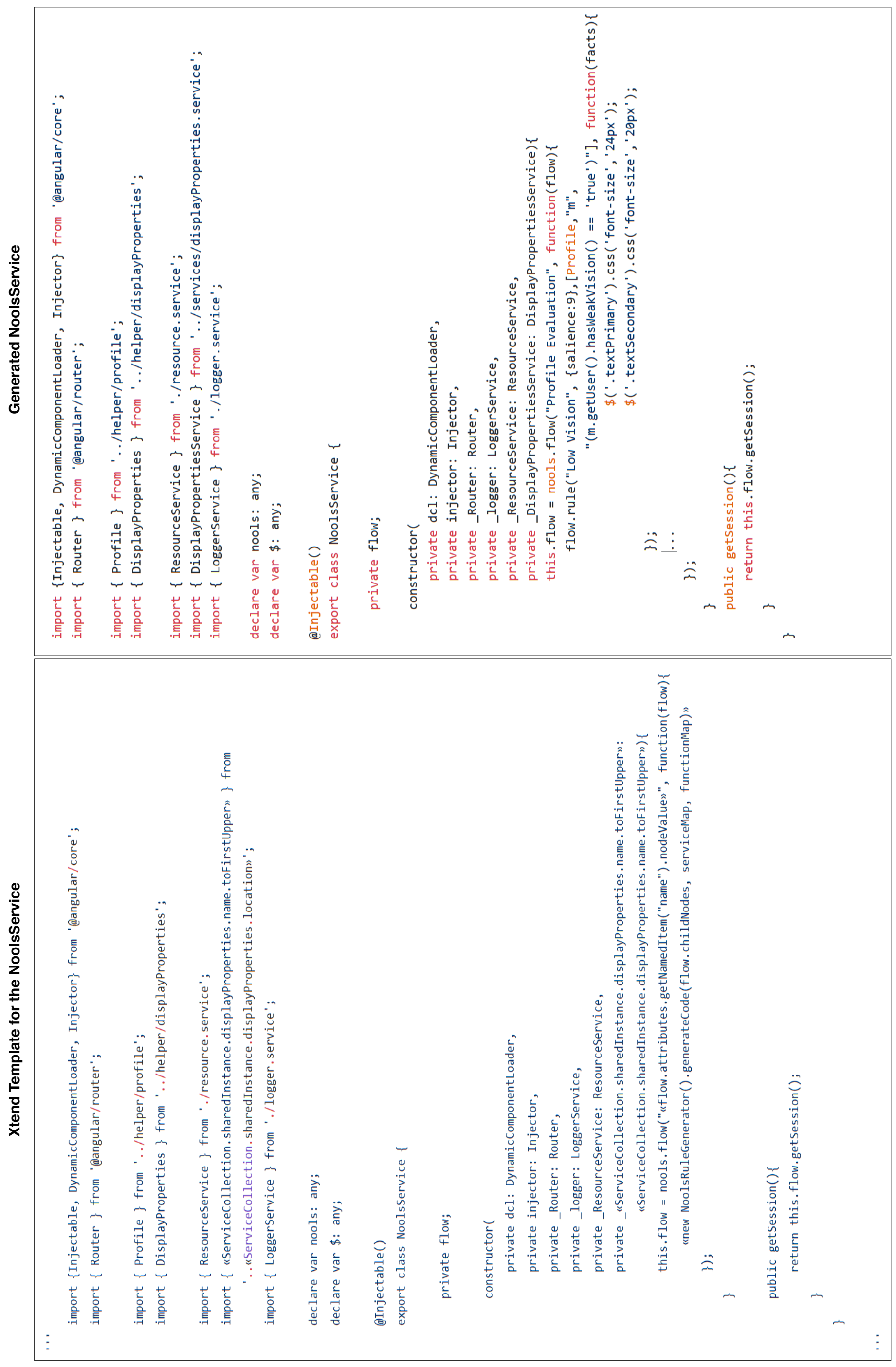
Code Excerpt 5.7 shows the Xtend emplate for the *NoolsConditionGenerator*. The *NoolsConditionGenerator* is responsible for creating the rule conditions. All child elements of the condition element are combined with the OR-operator. If there is a *conditionGroup* element, all child elements of the *conditionGroup* are combined with the AND-operator. The result is a string of concatenated conditions with operators.

```
public class NoolsConditionGenerator extends AbstractViewElementGenerator<Premise>{
    ...
    def String recursiveConditionUnrolling(Condition c){
        val output = new LinkedList<String>;
        if(c instanceof AndCombinedCondition){
            for(subCondition:c.subConditions){
                output.add(recursiveConditionUnrolling(subCondition));
            }
            return output.join(’ && m.’);
        }
        else if(c instanceof OrCombinedCondition){
            for(subCondition:c.subConditions){
                output.add(recursiveConditionUnrolling(subCondition));
            }
            output.join(’ || m.’);
        }
        else if(c instanceof PrimeCondition){
            return c.conditionAttribute + ’()’ + c.operator + ’’ + c.value +’’
        }
    }
    ...
}
```

Code Excerpt 5.7 Xtend template excerpt from *NoolsConditionGenerator*

Likewise, to generate the actions that the rule should execute when the conditions are satisfied, the *NoolsActionGenerator* is called with the actions element as parameter. As this is characterized mainly by a parameter handover, we rather show an illustrative example instead of the Xtend template.

An example for the generation of a *Nools Adaptation Service* is depicted in Figure 5.19. The code excerpt depicted in Figure 5.19 represents on the left side the Xtend template for generating a *Nools Adaptation Service*. On the right side of Figure 5.19, the generated code for the *Nools Adaptation Service* is illustrated, which is at runtime responsible for executing the UI adaptations.


Xtend Template for the NoolsService

```
...
    import {Injectable, DynamicComponentLoader, Injector} from '@angular/core';
    import { Router } from '@angular/router';

    import { Profile } from '../helper/profile';
    import { DisplayProperties } from '../helper/displayProperties';

    import { ResourceService } from './resource.service';
    import { «ServiceCollection.sharedInstance.displayProperties.name.toFirstUpper» } from
          '..«ServiceCollection.sharedInstance.displayProperties.location»';
    import { LoggerService } from './logger.service';

    declare var nools: any;
    declare var $: any;

    @Injectable()
    export class NoolsService {

        private flow;

        constructor(
            private dcl: DynamicComponentLoader,
            private injector: Injector,
            private _Router: Router,
            private _logger: LoggerService,
            private _ResourceService: ResourceService,
            private _«ServiceCollection.sharedInstance.displayProperties.name.toFirstUpper»:
                     «ServiceCollection.sharedInstance.displayProperties.name.toFirstUpper»){
            this.flow = nools.flow("«flow.attributes.getNamedItem("name").nodeValue»", function(flow){
                «new NoolsRuleGenerator().generateCode(flow.childNodes, serviceMap, functionMap)»
            });
        }

        public getSession(){
            return this.flow.getSession();
        }
    }

...
```

Generated NoolsService

```
import {Injectable, DynamicComponentLoader, Injector} from '@angular/core';
import { Router } from '@angular/router';

import { Profile } from '../helper/profile';
import { DisplayProperties } from '../helper/displayProperties';

import { ResourceService } from './resource.service';
import { DisplayPropertiesService } from '../services/displayProperties.service';
import { LoggerService } from './logger.service';

declare var nools: any;
declare var $: any;

@Injectable()
export class NoolsService {

    private flow;

    constructor(
        private dcl: DynamicComponentLoader,
        private injector: Injector,
        private _Router: Router,
        private _logger: LoggerService,
        private _ResourceService: ResourceService,
        private _DisplayPropertiesService: DisplayPropertiesService){
        this.flow = nools.flow("Profile Evaluation", function(flow){
            flow.rule("Low Vision", {salience:9},[Profile,"m",
                          "(m.getUser().hasWeakVision() == 'true')"], function(facts){
                              $('.textPrimary').css('font-size','24px');
                              $('.textSecondary').css('font-size','20px');
            });
            |...
        });
    }
    public getSession(){
        return this.flow.getSession();
    }
}
```

Fig. 5.19 Xtend template excerpt for *NoolsServiceGenerator* and its generated code

5.5 Summary and Discussion

In this chapter, we have presented our transformation approach for self-adaptive user interfaces (SAUIs). Its main goal is to support the generation of code for the *Final UI (FUI)*, *Context Services*, and *Adaptation Services* in an automated way. For this purpose, we gave an overview of the transformation approach and specified mappings between the source models and target artifacts to enable the implementation of template-based code generators. Based on the specified mappings characterized through Xtend templates, the implementation of the *SAUI-Generator* was explained in more detail by elaborating on each specific subgenerator *UI Generator*, *Context Service Generator*, and *Adaptation Service Generator*. In summary, the implemented *SAUI-Generator* supports the development approach of SAUIs. Based on the specified models (*Domain Model*, *IFML*, *ContextML*, and *AdaptML*), the creation of the *FUI*, *Context Service* and *Adaptation Service* in supported in an automated way.

While the implemented *SAUI-Generator* illustrates the main idea of realizing model-driven SAUI development based on the Angular framework focusing on web technologies, it is also conceivable to use our approach to devise SAUIs for other target technologies. In this case, our Xtend templates can be reused and adjusted to realize the requirements for the new target platform. As our proof-of-concept implementation of the *SAUI-Generator* supports a subset of *IFML* model elements, it is also possible to extend our template-based code generator with further template expressions to cover more IFML model elements in order to realize more sophisticated UIs. Similarly, the existing set of Xtend templates for generating *Context* and *Adaptation Services* can be extended to cover further context monitoring and UI adaptation aspects.

Regarding the generation of *Context Services* it should be marked that due to the very individual structure of some context sources, the code for controlling and managing sensor sources, like APIs, SDKs or libraries, could not be always automatically generated and thus had to be implemented manually. The code is written in static files, which are scanned during the generation process and inserted into predefined sections among the generated code using the concept of protected regions. Hence, the proof-of-concept implementation of the *Context Service Generator* covers the automated generation of code for a specific set of *Context Services* and can be extended to cover further context sensors and information.

Our *Adaptation Service* generation approach relies on the usage of *Nools* which is an efficient rule engine based on the RETE algorithm. Instead of *Nools* it is conceivable to use other rule engines such as Drools[10] or NRules[11]. In such a case, the flexible structure of our *Adaptation Service Generator* can be easily adjusted according to new code templates to enable a translation of the *AdaptML* models to the syntax of the target rule engine.

[10]https://www.drools.org
[11]https://github.com/NRules/NRules

Chapter 6

Execution

In this chapter, we present the execution environment for self-adaptive UIs. Firstly, in Section 6.1, we describe the runtime architecture for supporting context monitoring and UI adaptation. After that, in Section 6.2, we describe the proof-of-concept implementation of our rule-based execution environment for realizing self-adaptive UIs. Subsequently, in Section 6.3, we present our tool-chain which was implemented to support our model-driven development approach for self-adaptive UIs. Finally, main results of this chapter are briefly summarized and discussed in Section 6.4.

6.1 Runtime Architecture for Self-adaptive UIs

In the previous two chapters, we presented the modeling and transformation phases of our model-driven development approach for self-adaptive UIs. For enabling the execution of self-adaptive UIs, the following main requirements were identified in Section 3.2:

- **R3 - Runtime Monitoring**: The approach should support runtime context monitoring in such a way that the generated *Context Services* continuously observe context information and detect context changes through corresponding hardware sensors. The resulting *Context Services* shall provide context information to the UI *Adaptation Service* via data interface.
- **R7 - Runtime Adaptation**: The approach should support runtime UI adaptation in such a way that the generated *Adaptation Services* enable to automatically change the UI as a reaction to context changes.

To address the above mentioned requirements, existing Web and UI frameworks are not sufficient as they do not explicitly cover context management and UI adaptation concerns in an integrated execution framework. Usually, classical Web and UI frameworks rather support a generic execution framework, commonly based on the Model View Controller (MVC)[1] paradigm (or similar paradigms), to enable the view representation, data binding, and integration of application logic services. As the focus of this thesis is to support model-driven development of self-adaptive UIs in a flexible and modularized way, a novel execution environment for self-adaptive UIs is required that fulfills the before mentioned requirements.

For this purpose, we have conceptualized a generic runtime architecture for self-adaptive UIs which is shown in Figure 6.1.

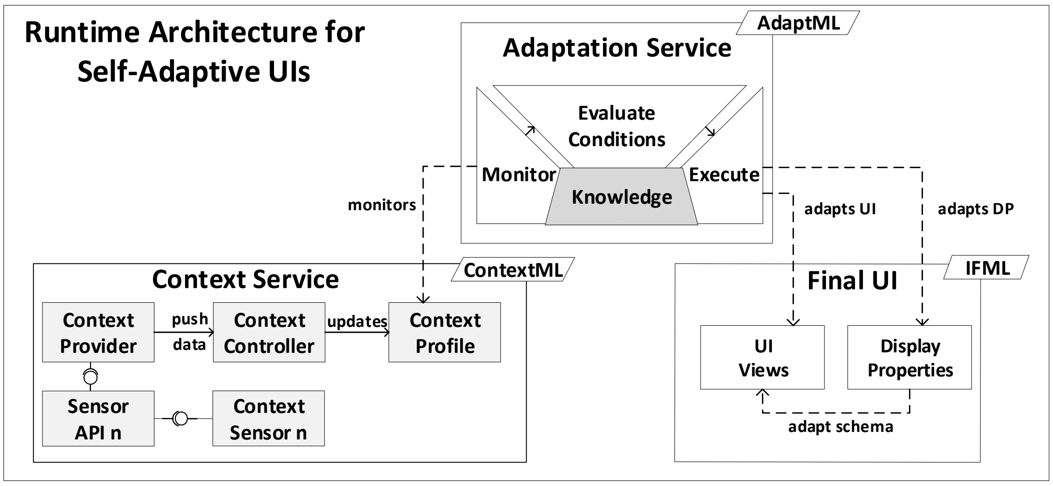


Fig. 6.1 Overview of Conceptual Runtime Architecture for Self-adaptive UIs

The depicted runtime architecture consists of three main components, namely, *Final UI*, *Context Service*, and *Adaptation Service* which are integrated in an overall execution environment.

The *Final UI*, which is derived based on the specification of the *IFML* model, has different subcomponents to enable the view representation based on specific *UI Views* and *Display Properties*. As this component supports basic functions for displaying and controlling the UI elements, it can be covered through a classical Web or UI framework as described in the beginning of this section.

To extend the UI framework in order to support context monitoring and UI adaptation at runtime as well as to support an overall execution environment for self-adaptive UIs, the

[1]https://martinfowler.com/eaaDev/uiArchs.html

following components were added and integrated into the runtime architecture:

The *Context Service* is derived based on the specification of the *ContextML* model and has different subcomponents to provide context information to the UI adaptation component (*Adaptation Service*) via data interface. As already introduced in the *ContextML* modeling Section 4.3, the subcomponent *Context Provider* is essential for gathering context information data through *Context Sensors*. To collect various context information, the *Context Sensors* are accessed through a dedicated *Sensor API*. At runtime, the *Context Service* uses the *Context Provider* to continuously monitor various context-of-use information. To provide the needed context information data, the *Context Provider* pushes the context information data to the *Context Controller* which again updates the *Context Profile* after each context change. The *Context Profile* contains the current context-of-use information about the sensed contextual parameters and delivers it through a data interface.

The *Adaptation Service*, which is derived based on the specification of the *AdaptML* model, is responsible for enabling runtime UI adaptation. It is based on IBM's MAPE-K loop [IBM05] and consists of the following subcomponents *Monitor*, *Evaluate Conditions*, *Execute*, and *Knowledge*. As the name implies, the *Monitor* subcomponent monitors the context informtion data that is delivered by the *Context Profile* subcomponent of the *Context Service*. The context information data is processed in the *Evaluate Conditions* subcomponent which analyzes and plans the execution of the UI adaptation operations as they were specified based on the UI adaptation rules specified with the help of *AdaptML*. The *Execute* subcomponent is finally responsible for applying the UI adaptation operations on the *Final UI*. Executed UI adaptation operations can effect changes to specific *UI Views* or trigger more generic changes by changing the *Display Properties*. Finally, the *Knowledge* subcomponent is responsible for storing context information and applied UI adaptation rules. In the next chapter, it will be explained how also user feedback can be logged in the *Knowledge* subcomponent to enable usability evaluation of UI adaptations.

In the following section, we present how the sketched runtime architecture has been implemented as a proof-of-concept execution environment for supporting self-adaptive UIs.

6.2 Execution Environment for Self-adaptive UIs

In order to realize the idea of a runtime architecture for self-adaptive UIs, we implemented a rule-based execution environment for self-adaptive UIs. Our proof-of-concept implementation of the execution environment relies on the open source Angular framework[2] which is one of the most used UI/Web frameworks. Angular is maintained by Google and aims to facilitate the development of modern UIs and (web) applications for different target platforms by relying on established development practices, concepts, and conventions. Angular has established itself as a de facto standard for front-end/UI development purposes and is highly used in industrial projects. As already introduced in section 2.6 and recaptured in Figure 6.2, the architecture of the Angular framework is very modular.

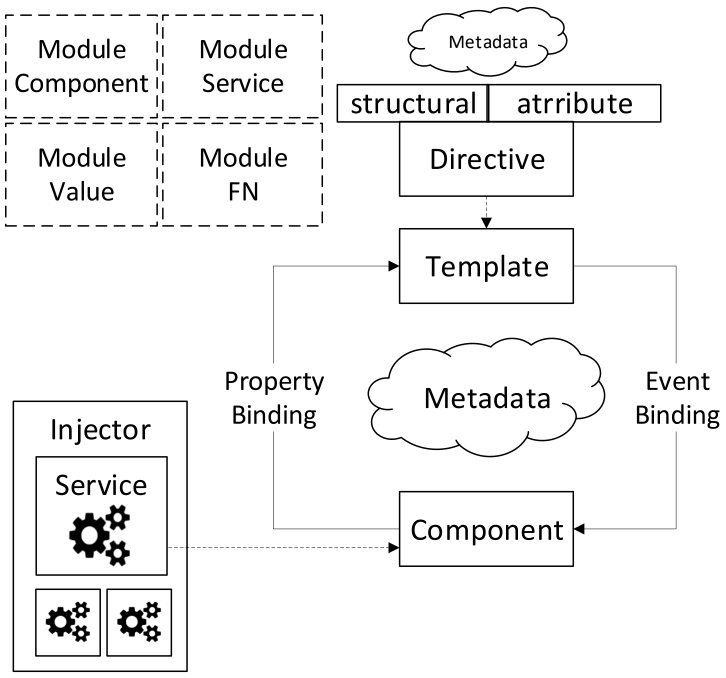


Fig. 6.2 Architectural Overview of the Angular framework [GS13]

Beside that, the architecture of the Angular framework fits well to to our concern to characterize the UI and reflect UI changes to realize runtime UI adaptation. Moreover, the modularity of the Angular architecture supports sufficient flexibility to further integrate *Context Services* and *Adaptation Services* to cover our concerns context monitoring and UI adaptation at runtime. Therefore, the Angular framework has been chosen as a basis UI framework and extended for the proof-of-concept implementation of our execution environment for self-adaptive UIs, which is depicted in Figure 6.3.

Our execution environment uses the existing Angular concepts for characterizing the *Final UI* which is generated by the sub-generator *UI Generator* (see Section 5.2). The *Final UI*

[2]https://angular.io

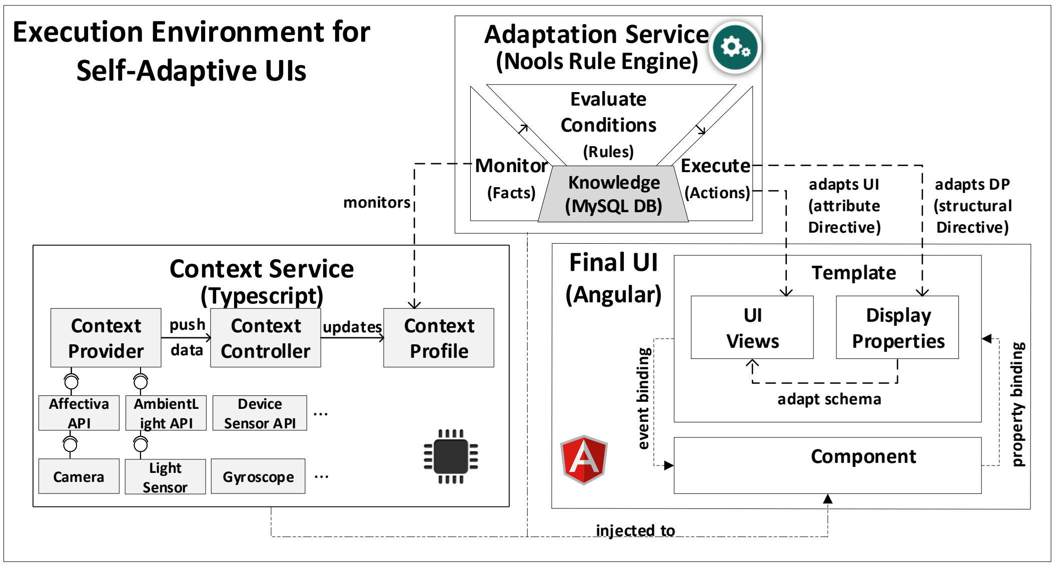


Fig. 6.3 Overview of implemented Execution Environment for Self-adaptive UIs

consists of *Templates* describing the *UI Views* and *Display Properties* in HTML as well as *Components* in form of *Typescript* code for managing the *UI Views* and binding functional services to cover business logic.

Beside the *Final UI*, our execution environment consists of *Context Services* that are generated by the sub-generator *Context Service Generator* (see Section 5.3) and represented in Typescript code. For integrating *Context Services* into our execution environment, we use Angular's *Injector* component which supports code injection. At runtime, the generated *Context Service* works as a background service, that can be used by any application based on the *Angular* framework. For this purpose, the *Context Provider* accesses the context information data through different *Sensor APIs* (e.g., Affectiva API, Light Sensor API, Device Sensor API, etc.) by using the hardware *Context Sensors* (e.g., Camera, Light Sensor, Gyroscope, etc.). The gathered context information data is forwarded to the *Context Controller*. Through the subjects of the observer pattern, new data is directly pushed to the subscriptions of the *Context Controller*. At the same time, the corresponding property is updated in the *Context Profile* which can be accessed to get the current-context-of use information.

Apart from the previously described components, our execution environment for self-adaptive UIs needs a component to realize the adaptation logic for enabling UI adaptation at runtime. For this purpose, the rule engine *Nools* is integrated into our execution environment by using the code injection technique. *Nools* is an efficient Javascript based rule engine which is based on the *RETE* algorithm [For82]. Due to its many advantages such as flexibility, reusability, efficiency etc. (see Section 2.6 for further details) and to reduce the complexity in handling UI adaptations at runtime the *Nools* rule engine is an integral part of our execution environment.

In this thesis, *Nools* is integrated into the execution environment to support the execution of the *Adaptation Service* and to enable runtime UI adaptation. The *Adaptation Service* uses *Nools* for monitoring the context information provided by the *Context Service* and executes the adaptation rules whose conditions are satisfied. To adapt the *UI View* elements of the *Final UI* on instance level, JQuery[3] is used to directly manipulate the DOM tree of the *UI View*. Changes only affect the current *UI View* element and do not persist in other *UI views*. When changing the schema for a group of *UI View* elements in the *Display Properties*, the adaptation affects the properties of all *UI View* elements of this type. This also includes instances of this *UI View* element type on subsequently visited *UI Views*. This is done by binding the layout class of the *UI View* elements of this type, represented by CSS classes, to the properties stored within the *Display Properties*.

In summary, our described execution environment extends the existing Angular UI framework to support context management and UI adaptation concerns by integrating the *Nools* rule engine. The *Nools* rule engine enables to manage the *Context Services* and *Adaptation Services*. Thus, runtime context monitoring and UI adaptation is supported in an integrated execution environment that was protoypically implemented.

6.3 Tool-Support: Adapt-UI IDE

In this section, we present our integrated development environment (IDE) for supporting model-driven development of self-adaptive UIs. This IDE, named *Adapt-UI*, provides integrated views for UI, context and adaptation modeling. Based on the specified models, *Final UI* code and *Context* as well as *Adaptation Services* are generated and integrated in an overall UI framework. This allows runtime UI adaptation realized by an automatic reaction to context-of-use changes.

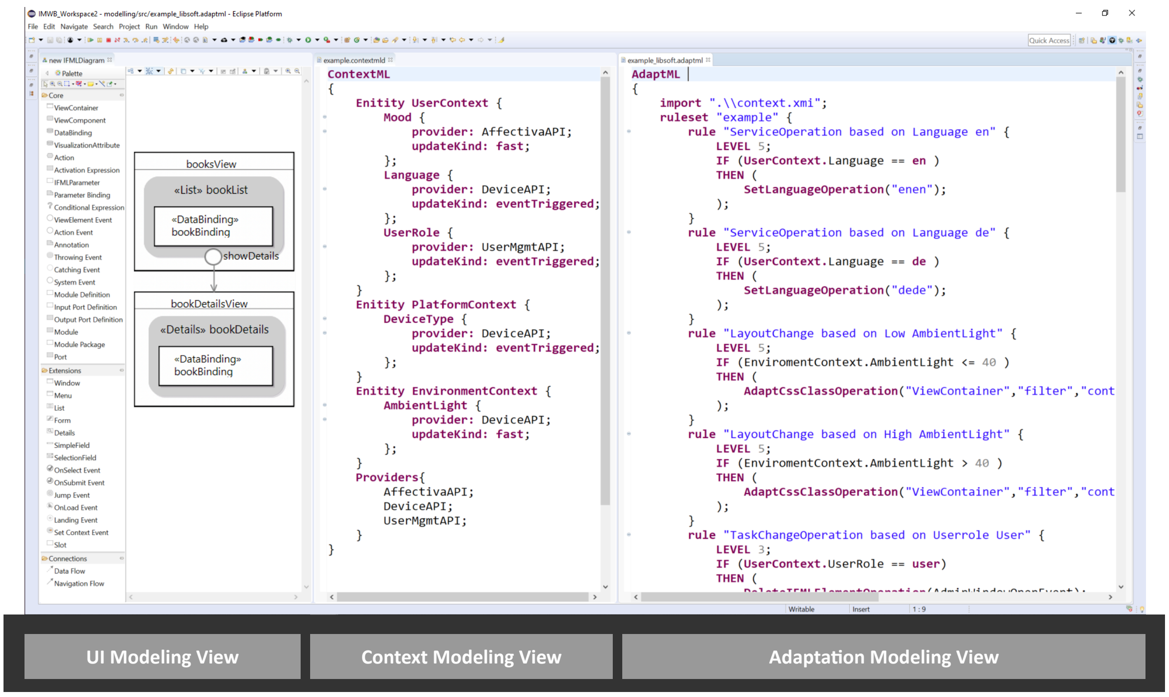
In the following sections, the main features of *Adapt-UI* and the process of using this tool to support model-driven development of self-adaptive UIs is explained. Therefore, the

[3]https://jquery.com

implemented modeling workbench of *Adapt-UI* and the provided generation tools are briefly summarized.

6.3.1 Modeling Workbench

The modeling workbench of Adapt-UI (see Figure 6.4) provides three different modeling views to support model-driven development of self-adaptive UIs: UI, context, and adaptation. In the following, each view is shortly explained.



```
ContextML
{
    Enitity UserContext {
        Mood {
            provider: AffectivaAPI;
            updateKind: fast;
        };
        Language {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        UserRole {
            provider: UserMgmtAPI;
            updateKind: eventTriggered;
        };
    }
    Enitity PlatformContext {
        DeviceType {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
    }
    Enitity EnvironmentContext {
        AmbientLight {
            provider: DeviceAPI;
            updateKind: fast;
        };
    }
    Providers{
        AffectivaAPI;
        DeviceAPI;
        UserMgmtAPI;
    }
}
```

```
AdaptML
{
    import ".\\context.xmi";
    ruleset "example" {
        rule "ServiceOperation based on Language en" {
            LEVEL 5;
            IF (UserContext.Language == en )
            THEN (
                SetLanguageOperation("enen");
            );
        }
        rule "ServiceOperation based on Language de" {
            LEVEL 5;
            IF (UserContext.Language == de )
            THEN (
                SetLanguageOperation("dede");
            );
        }
        rule "LayoutChange based on Low AmbientLight" {
            LEVEL 5;
            IF (EnviromentContext.AmbientLight <= 40 )
            THEN (
                AdaptCssClassOperation("ViewContainer","filter","cont
            );
        }
        rule "LayoutChange based on High AmbientLight" {
            LEVEL 5;
            IF (EnviromentContext.AmbientLight > 40 )
            THEN (
                AdaptCssClassOperation("ViewContainer","filter","cont
            );
        }
        rule "TaskChangeOperation based on Userrole User" {
            LEVEL 3;
            IF (UserContext.UserRole == user)
            THEN (
```

Fig. 6.4 Adapt-UI Development Environment

UI Modeling View: The goal of the *UI Modeling View* is to support the specification of an abstract UI model that serves as a basis for generating the final UI. For accomplishing this task, Adapt-UI makes use of the *Interaction Flow Modeling Language (IFML)*, which is standardized by the Object Management Group (OMG). Based on the open source *IFML Editor Eclipse Plugin* that is integrated into our IDE, developers are able to specify core UI aspects. That means, a domain model describing the relevant data entities of the UI and an *IFML* model characterizing the structure, content and navigation of the UI can be modeled in a graphical notation.

Context Modeling View: Beside the UI model, the modeling workbench of *Adapt-UI* provides a *Context Modeling View*, which enables to specify the context model for characterizing dynamically changing context-of-use parameters. The *Context Modeling View* is based on our context modeling language *ContextML* and covers the main context entities user, platform, and environment, which are characterized by various attributes. Beside the context entities, the context modeling view also enables the definition of context providers that are used for sensing special context information data. Based on *ContextML* different potential contextual parameters can be modeled, so that a customizable context manager is generated for different usage scenarios.

Adaptation Modeling View: Moreover, the *Adapt-UI* modeling workbench enables the specification of UI adaptation rules based on our adaptation modeling language *AdaptML*. *AdaptML* is based on the Event-Condition-Action (ECA) Paradigm and supports the different categories of UI adaptation techniques such as task change operation, navigation change operation, layout change operation, and modality change operation. Task change operations support UI adaptation by flexibly showing and hiding UI interaction elements like tables, buttons, text-fields etc. Navigation change operation means that the navigation flow of the UI can be flexibly adapted based on the contextual parameters by adding, deleting or redirecting links between user interface flows. Furthermore, layout change operation deals with adaptation rules that support layout optimization like changing font size, colors or positioning. Finally, a modality change operation enables to switch the UI's interaction modality, for instance between graphical or vocal mode. As it is shown in the adaptation modeling view (see Figure 2), *AdaptML* references context entity attributes like *age* or *mood* to define application conditions for the UI adaptation rules. *AdaptML* also allows specifying and binding different adaptation rules to the *IFML* modeling elements, because the adaptation modeling view is linked with both views context and UI modeling.

6.3.2 Code Generators

The integrated development environment *Adapt-UI* provides the following core code generators:

UI Generator: The provided *UI Generator* is responsible for automatically generating the *Final UI* code based on the *IFML* specification. The *Final UI* code consists of Angular views which are represented as an HTML template to display the UI in the browser, and an Angular component, which is implemented in TypeScript and manages the view.

Context Service Generator: The goal of the provided *Context Service Generator (CSG)* is the automated creation of Angular services for sensing and providing context sensory data for triggering the adaptation mechanism. Therefore, based on the specified context model with *ContextML*, the CSG automatically creates code for the *Context Services* that themselves make use of existing libraries like Affectiva API or Geolocation API. The CSG is implemented with Xtend and realizes a model-to-text transformation where context model elements are mapped to corresponding *Context Service* function calls of the used context sensor libraries.

Adaptation Service Generator: The goal of the *Adaptation Service Generator* (ASG) is the automated creation of an Angular service that allows the adaptation of the UI at runtime. The adaptations to the UI are expressed in a rule-based form based on *AdaptML*. Based on this input file, the ASG generates an Angular service containing the JavaScript rule engine *Nools*. *Nools* is an efficient RETE-based rule engine written in JavaScript and provides an API for specifying facts and rules. The ASG is also implemented with Xtend and receives the UI adaptation rules as input. Based on this input, the ASG is responsible for creating an injectable Angular service for monitoring the context information and executing UI adaptation operations.

6.4 Summary and Discussion

In this chapter, we have presented a runtime architecture for self-adaptive UIs which extends classical UI frameworks through context management and UI adaptation concerns. As a proof-of-concept implementation of the runtime architecture, we have implemented an execution environment based on the Angular framework that was extended through a rule engine to enable context monitoring and UI adaptation at runtime. Furthermore, we have presented our integrated development environment *Adapt-UI* which builds around a modeling workbench and the provided code generators to derive self-adaptive UIs.

Regarding the execution of self-adaptive UIs, it should be noticed that the execution of UI adaptations relies on the specification of the *AdaptML* model. Due to human errors it is possible, that conflicting UI adaptations may be specified that lead to unexpected behavior of the used interactive system. With this regard, further quality assurance mechanisms to ensure a reliable execution of self-adaptive UIs are needed. In this context, it is conceivable, for example, to use model-checking techniques to verify the correctness of UI adaptation operations.

Chapter 7

Evaluation

In the previous chapters, we presented our model-driven development part of our engineering approach for self-adaptive user interfaces. This chapter deals with the evaluation part of our model-driven engineering approach consisting of two parts: an analysis of the applicability of our model-driven development approach based on two case studies as well as a usability evaluation of the resulting self-adaptive UIs.

Firstly, in Section 7.1, the benefit of our model-driven development approach is demonstrated by two case-studies showing the development of self-adaptive UIs for different application scenarios. The first application scenario for which we devised self-adaptive UIs is a library web application. The second application scenario deals with the development of an e-mail application with UI adaptation capabilities. In Section 7.2, we present our usability evaluation for self-adaptive UIs. For this purpose, we introduce our novel usability evaluation solution for on-the-fly usability testing of self-adaptive UIs. Furthermore, we report on a usability study that was conducted based on this and present its results. Moreover, we discuss the results and limitations of the usability evaluation. Finally, Section 7.3 concludes this chapter by summarizing and discussing the main findings of the evaluation.

7.1 Case Studies

The purpose of the cases studies is to evaluate the applicability of the solution concept of this thesis, i.e., the model-driven development approach for self-adaptive UIs. Therefore, two different real-world application scenarios were selected to instantiate our development approach and to analyze its benefit regarding practice of use. For the case studies, we aim to evaluate the applicability of our solution approach with respect to the following questions that were derived based on the identified requirements in Section 3.2:

EQ1 Does the solution approach support the integrated modeling of self-adaptive UIs by covering relevant concerns such as core UI aspects, context management, and UI adaptation?

EQ2 Does the solution enable the generation of code for the *Final UI*, *Context Services*, and *Adaptation Services* to automate the development approach for self-adaptive UIs?

EQ3 Does the solution approach enable runtime UI adaptation by integrating the generated artifacts *Final UI* code, *Context*, and *Adaptation* Services in an overall rule-based execution environment?

In the following, we present the case studies and discuss for each application scenario the previously described evaluation questions.

7.1.1 Case-Study 1: Library Application (*LibSoft*)

The first case study has already been introduced in Section 3.1 as a running example. This example is derived from the library management domain (see Figure 7.1). The scenario setting is a library web application for universities called *LibSoft*.

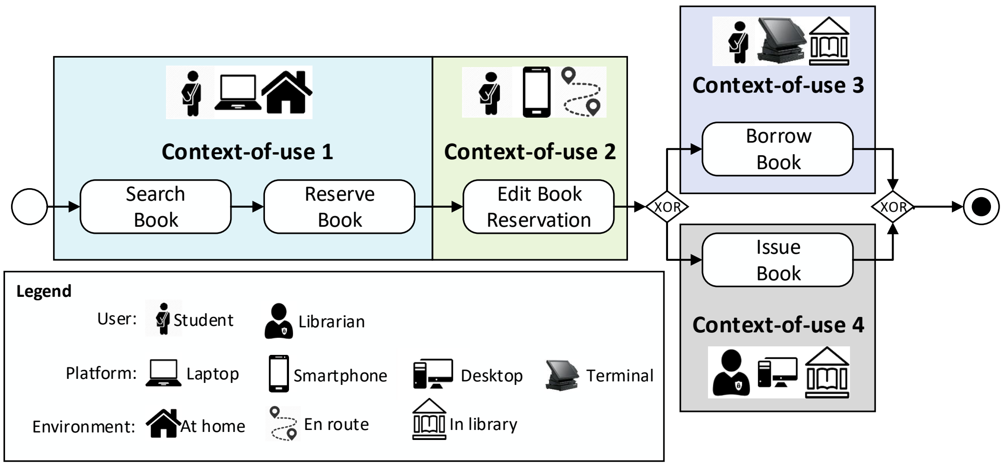


Fig. 7.1 Example scenario: UIs in dynamically changing context-of-use situations

LibSoft provides core library management functionality like searching, reserving, and borrowing books. *LibSoft*'s UI can be accessed by heterogeneous users and user roles (like student or staff member) through a broad range of networked interaction devices (e.g., smartphones, tablets, terminals etc.) used in various environmental contexts (e.g., brightness, loudness, while moving etc.). Depending on the situation, users are able to access their library services where, when, and how it suits them best. For example, if the user wants to pursue a self-determined cross-channel book borrowing process, she can begin an interaction using one channel (search and reserve a book with her laptop at home), modify the book reservation on her way using a mobile channel, and finalize the book borrowing process at the library via self-check-out terminal or at the staff desk. In the above described example scenario, each channel has its own special context-of-use and the contextual parameters regarding user, platform, and environment can dynamically change. Figure 7.2 shows a concrete context-of-use (*CoU*) change from *CoU2* to *CoU4* (compare Figure 7.1). The depicted context-of-use object model excerpts in Figure 7.2 illustrate how different contextual parameters regarding user, platform, and environment change. Already a small set of contextual parameters can highly influence the usability of the UI as many context-of-use parameters might dynamically change. Therefore, it is important to continuously monitor the context-of-use parameters and react to possible changes by automatically adapting the UI for the new context-of-use situation.

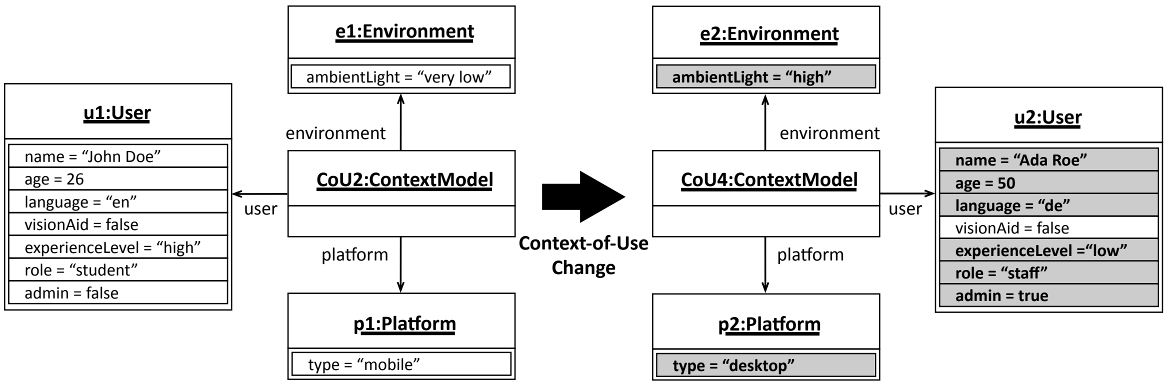


Fig. 7.2 Library application: Context-of-use object model excerpts

For utilizing our integrated model-driven development approach in the case study setting, a *Domain Model*, an *IFML Model*, a *Context Model*, and an *Adaptation Model* with a set of UI adaptation rules were created as described in Chapter 4. Figure 7.3 depicts the domain model of the *LibSoft* application in form of a UML class diagram specifying the main data entities for the user interface. An *IFML* model excerpt specifying example UI views of the *LibSoft* application is depicted in Figure 7.4.

It shows mainly four UI views characterizing the abstract UI for the login page specified as *loginForm*, home menu denoted as *home*, and the pages for searching for books *searchBooks* and reserving books *bookReservations*. The *loginForm* is simply describing the login form for entering username and password. The *home ViewContainer* contains a *ViewComponent*

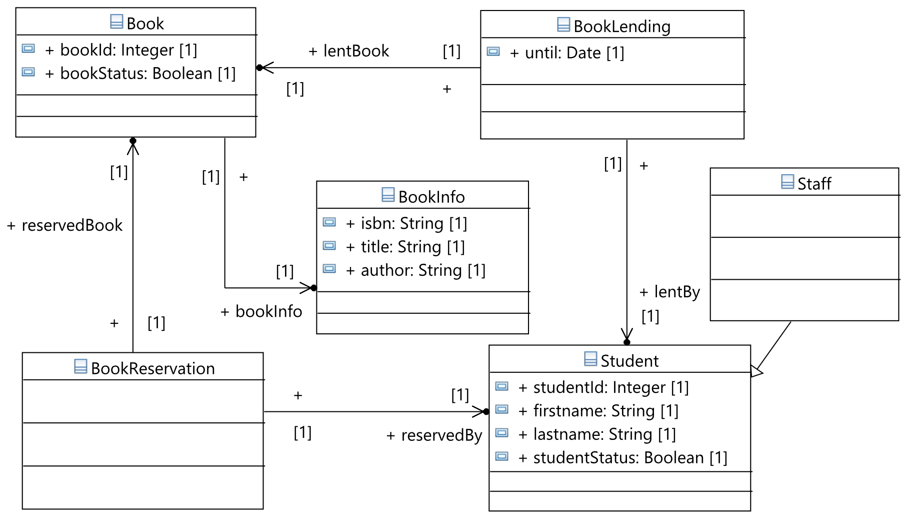


Fig. 7.3 Domain model for *LibSoft* example scenario

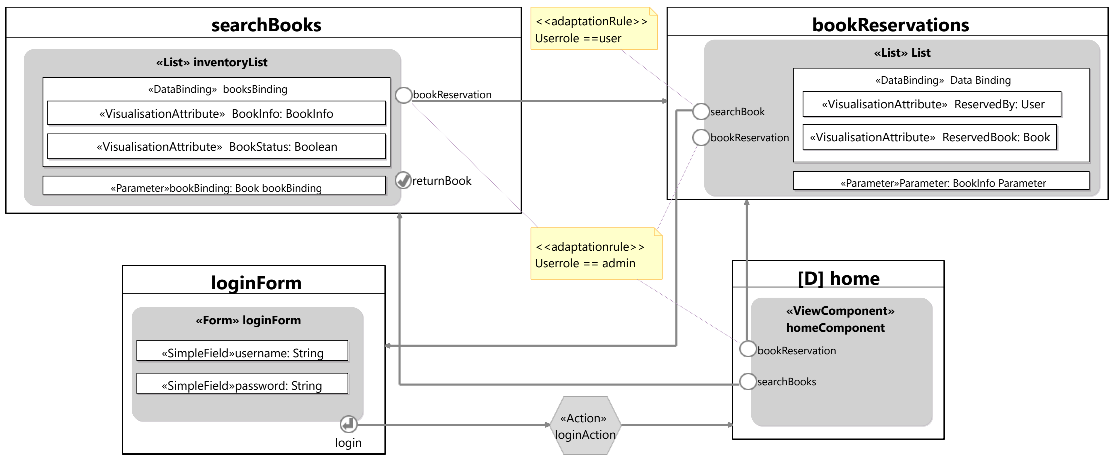
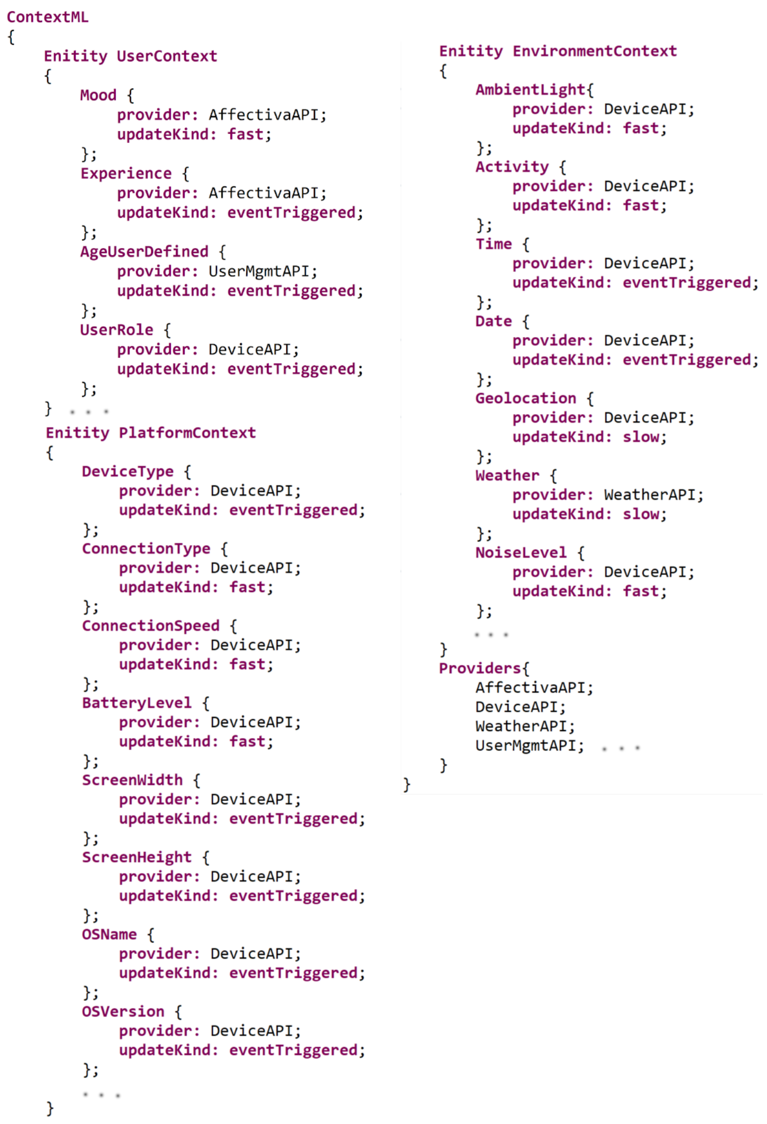

searchBooks
«List» inventoryList
«DataBinding» booksBinding
«VisualisationAttribute» BookInfo: BookInfo
«VisualisationAttribute» BookStatus: Boolean
«Parameter»bookBinding: Book bookBinding
bookReservation
returnBook
<<adaptationRule>>
Userrole ==user
bookReservations
«List» List
«DataBinding» Data Binding
«VisualisationAttribute» ReservedBy: User
«VisualisationAttribute» ReservedBook: Book
«Parameter»Parameter: BookInfo Parameter
searchBook
bookReservation
<<adaptationrule>>
Userrole == admin
loginForm
«Form» loginForm
«SimpleField»username: String
«SimpleField»password: String
login
«Action»
loginAction
[D] home
«ViewComponent»
homeComponent
bookReservation
searchBooks

Fig. 7.4 IFML model excerpt for *LibSoft* example scenario

called *homeComponent* which characterizes the navigation menu. It is denoted with a [D] as it is the default landing page from where on different pages can be accessed. One of these reachable pages is for example *searchBooks*. This *IFML* model element contains a list called *inventoryList* which is responsible for listing the found books with information about *BookInfo* and *BookStatus*, etc.. A further reachable page is *bookReservations* which shows an overview of the reserved books corresponding to the lender. The annotations shown in yellow background color, characterize authentication expressions for handling the UI adaptations regarding different user roles. For example, by default, it is specified that the *searchBook* event in *bookReservations* can be only accessed by the user with *user role* equals *user*. The other events in the example are by default specified to be accessible by the admin user role.

Furthermore, Figure 7.5 depicts an illustrative context model specified for the *LibSoft* example scenario. It specifies the relevant context-of-use parameters that should be monitored while the self-adaptive UI is used.



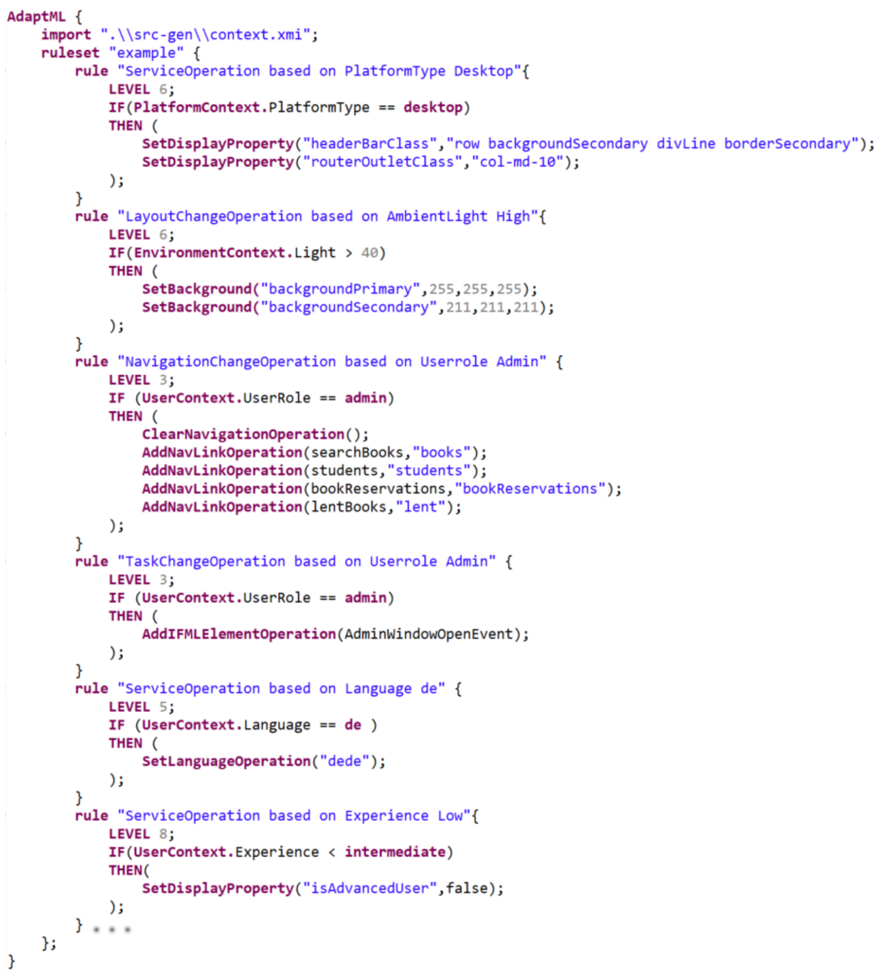
```
ContextML
{
    Enitity UserContext
    {
        Mood {
            provider: AffectivaAPI;
            updateKind: fast;
        };
        Experience {
            provider: AffectivaAPI;
            updateKind: eventTriggered;
        };
        AgeUserDefined {
            provider: UserMgmtAPI;
            updateKind: eventTriggered;
        };
        UserRole {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
    } . . .
    Enitity PlatformContext
    {
        DeviceType {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        ConnectionType {
            provider: DeviceAPI;
            updateKind: fast;
        };
        ConnectionSpeed {
            provider: DeviceAPI;
            updateKind: fast;
        };
        BatteryLevel {
            provider: DeviceAPI;
            updateKind: fast;
        };
        ScreenWidth {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        ScreenHeight {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        OSName {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        OSVersion {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        . . .
    }
    Enitity EnvironmentContext
    {
        AmbientLight{
            provider: DeviceAPI;
            updateKind: fast;
        };
        Activity {
            provider: DeviceAPI;
            updateKind: fast;
        };
        Time {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        Date {
            provider: DeviceAPI;
            updateKind: eventTriggered;
        };
        Geolocation {
            provider: DeviceAPI;
            updateKind: slow;
        };
        Weather {
            provider: WeatherAPI;
            updateKind: slow;
        };
        NoiseLevel {
            provider: DeviceAPI;
            updateKind: fast;
        };
        . . .
    }
    Providers{
        AffectivaAPI;
        DeviceAPI;
        WeatherAPI;
        UserMgmtAPI; . . .
    }
}
```

Fig. 7.5 *ContextML* model specified for the *LibSoft* example scenario

In a similar way, based on our modeling workbench, the needed UI adaptation rules can be specified for the *LibSoft* example scenario. Figure 7.6 shows a small set of such UI adaptation rules that were specified based on *AdaptML*.

Using our *SAUI-Generator* the specified models were transformed to *Final UI* code including the generated code for *Context* and *Adaptation Services*.

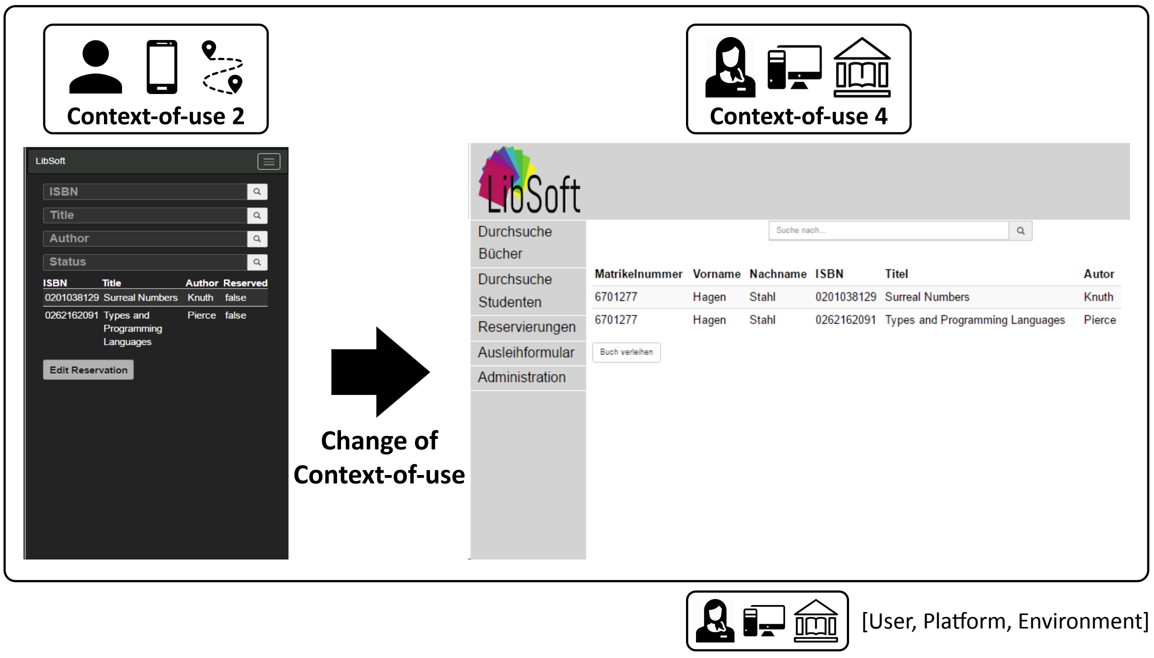
Example screenshots of the resulting self-adaptive UI are depicted in Figure 7.7. According to the monitored context information for *CoU2*, the layout for the UI is optimized



```
AdaptML {
    import ".\\src-gen\\context.xmi";
    ruleset "example" {
        rule "ServiceOperation based on PlatformType Desktop"{
            LEVEL 6;
            IF(PlatformContext.PlatformType == desktop)
            THEN (
                SetDisplayProperty("headerBarClass","row backgroundSecondary divLine borderSecondary");
                SetDisplayProperty("routerOutletClass","col-md-10");
            );
        }
        rule "LayoutChangeOperation based on AmbientLight High"{
            LEVEL 6;
            IF(EnvironmentContext.Light > 40)
            THEN (
                SetBackground("backgroundPrimary",255,255,255);
                SetBackground("backgroundSecondary",211,211,211);
            );
        }
        rule "NavigationChangeOperation based on Userrole Admin" {
            LEVEL 3;
            IF (UserContext.UserRole == admin)
            THEN (
                ClearNavigationOperation();
                AddNavLinkOperation(searchBooks,"books");
                AddNavLinkOperation(students,"students");
                AddNavLinkOperation(bookReservations,"bookReservations");
                AddNavLinkOperation(lentBooks,"lent");
            );
        }
        rule "TaskChangeOperation based on Userrole Admin" {
            LEVEL 3;
            IF (UserContext.UserRole == admin)
            THEN (
                AddIFMLElementOperation(AdminWindowOpenEvent);
            );
        }
        rule "ServiceOperation based on Language de" {
            LEVEL 5;
            IF (UserContext.Language == de )
            THEN (
                SetLanguageOperation("dede");
            );
        }
        rule "ServiceOperation based on Experience Low"{
            LEVEL 8;
            IF(UserContext.Experience < intermediate)
            THEN(
                SetDisplayProperty("isAdvancedUser",false);
            );
        } . . .
    };
}
```

Fig. 7.6 AdaptML model specified for the *LibSoft* example scenario

for a mobile device used in a darker environment, because the user John is editing his book reservation while travelling to the library and it is already quite dark outside (see left side of Figure 7.7). Also, the UI is adapted to the user properties by enabling access to the functions and navigation available to students. The UI language is set to *English* as it is preferred by John. As John is recognized as an experienced user with the application (based on his usage time), he gets extended functionalities, like a more complex search and filter mechanism for the list view of the books. When the context changes from *CoU2* to *CoU4*, the generated self-adaptive UI adapts itself automatically to the new contextual parameters. In this case, the staff members view on a desktop device with a wider and brighter layout is shown, displaying the list of reserved books, because in *CoU4* a staff member, Ada Roe, uses her desktop computer to issue the book to John. Additionally, to the functionalities and functions available to staff members, Ada is provided with a link to the administration interface, because she is granted access to the administration interface. The UI language is



ISBN	Title	Author	Reserved
0201038129	Surreal Numbers	Knuth	false
0262162091	Types and Programming Languages	Pierce	false

Matrikelnummer	Vorname	Nachname	ISBN	Titel	Autor
6701277	Hagen	Stahl	0201038129	Surreal Numbers	Knuth
6701277	Hagen	Stahl	0262162091	Types and Programming Languages	Pierce

Fig. 7.7 Library application: UI adaptations according to different contexts-of-use

set to German and the search and filter mechanisms of the list are simplified, because she just started using *LibSoft* and is, therefore, not yet experienced with the application. As the location is a well-lit library, the default brightness level is shown on the screen of the desktop computer.

The case study demonstrates the benefit of our approach for supporting the development of self-adaptive UIs. By using our integrated modeling workbench and the corresponding *SAUI-Generator*, we were able to model and generate a self-adaptive UI for the *LibSoft* example scenario. To sum up, our solution approach was helpful for addressing the introduced evaluation questions **EQ1-EQ3** as it provides an integrated model-driven development approach for self-adaptive UIs.

7.1.2 Case-Study 2: E-Mail Application (*MailSoft*)

As a second case study, we present a real world example scenario which is based on an e-mail application. E-mail applications are one of the most recurrently used applications on different devices. People read and write e-mails while commuting to work, before going to sleep, while walking, watching TV or doing different other activities. As various dynamically changing context-of-use situations are faced when using such an e-mail application, we decided to develop an e-mail application with UI adaptation capabilities, called *MailSoft*.

Therefore, we used again our integrated modeling environment for specifying the UI, context, and adaptation concerns of *MailSoft*.

Figure 7.8 depicts the domain model of the *MailSoft* application in form of a UML class diagram specifying the main data entities for the user interface.

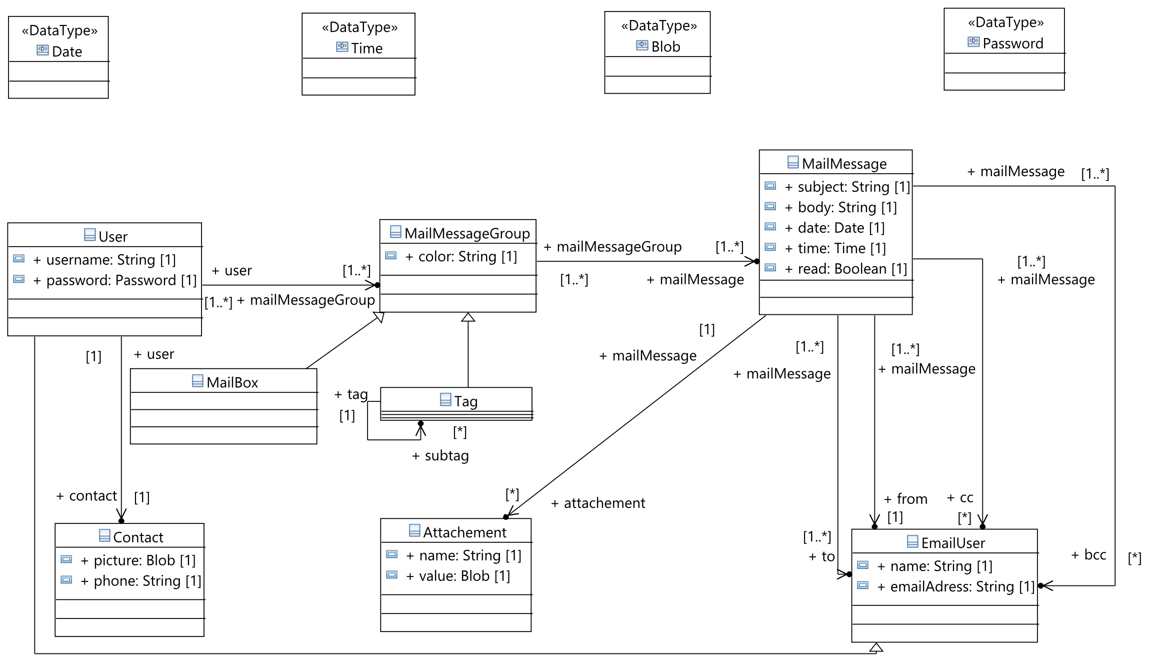


Fig. 7.8 Domain model for *MailSoft* example scenario

An *IFML* model excerpt specifying example UI views of the *MailSoft* application is depicted in Figure 7.9. It shows mainly two UI views characterizing the abstract UI for the main page specified as *homeViewInbox* and the page for reading mails denoted as *readViewReadMail*. The *homeViewInbox* view container describes the mail overview page where the binding and listing of mail messages are shown. By selecting a specific mail item, the navigation flow directs to a further view container which is called *readViewReadMail*. In this view container, detailed information about the mail message regarding sender, recipient, subject, and mail body are described. To overcome the limitations of *IFML*, also in this example scenario, we have used specific annotations (boxes with yellow background color) to better guide and support the UI generation. Firstly, the last mentioned visualisation attributes for specifying sender, recipient, subject, and mail body were annotated with an *order* value to describe the exact sequence of the elements for better layouting purposes. Secondly, the *mailMessageDetails* element is annotated with the label *readable* which denotes that the UI adaptation operation *SwitchUIModality(vocal)* (see Section 4.4) can be applied to this view element. Further annotations were used to encode functional aspects, for example, whether the search field is enabled or not. Finally, in some cases annotations were also used to encode important icon resources (*IconHolder* with path to icon source file).

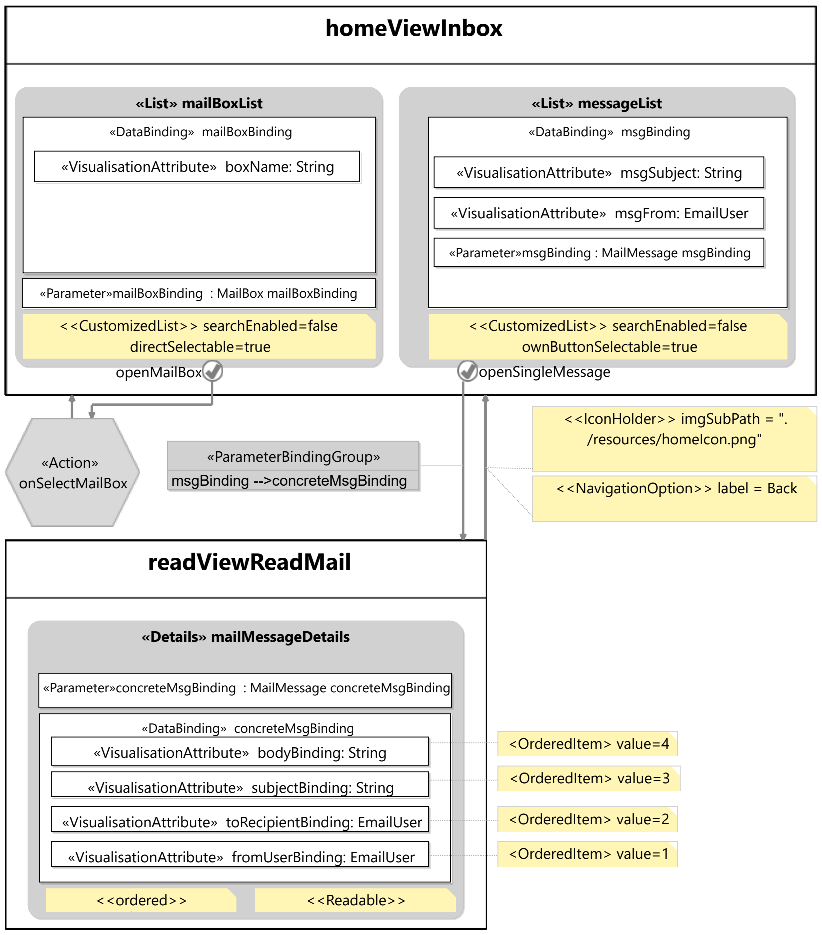

homeViewInbox
«List» mailBoxList
«DataBinding» mailBoxBinding
«VisualisationAttribute» boxName: String
«Parameter»mailBoxBinding : MailBox mailBoxBinding
<<CustomizedList>> searchEnabled=false directSelectable=true
openMailBox
«List» messageList
«DataBinding» msgBinding
«VisualisationAttribute» msgSubject: String
«VisualisationAttribute» msgFrom: EmailUser
«Parameter»msgBinding : MailMessage msgBinding
<<CustomizedList>> searchEnabled=false ownButtonSelectable=true
openSingleMessage
«Action» onSelectMailBox
«ParameterBindingGroup» msgBinding -->concreteMsgBinding
<<IconHolder>> imgSubPath = "./resources/homeIcon.png"
<<NavigationOption>> label = Back
readViewReadMail
«Details» mailMessageDetails
«Parameter»concreteMsgBinding : MailMessage concreteMsgBinding
«DataBinding» concreteMsgBinding
«VisualisationAttribute» bodyBinding: String
«VisualisationAttribute» subjectBinding: String
«VisualisationAttribute» toRecipientBinding: EmailUser
«VisualisationAttribute» fromUserBinding: EmailUser
<OrderedItem> value=4
<OrderedItem> value=3
<OrderedItem> value=2
<OrderedItem> value=1
<<ordered>>
<<Readable>>

Fig. 7.9 *IFML* model excerpt for *MailSoft* example scenario

The specified input models were given as input to our *SAUI-Generator* to generate code for the *Final UI*, *Context*, and *Adaptation Services* of the self-adaptive UI of the e-mail application.

Figure 7.10 depicts an illustrative sequence of context changes and how the UI of the e-mail application adapts to the changed context in each case.

Each state is a pair of the self-adaptive UI, depicted as a screenshot of the e-mail application, and the current context as experienced by the user. For the sake of simple visualization, the context is reduced to three components: (i) if the user is on the move, in a moving vehicle, or immobile (and probably at home), (ii) if the brightness level (ambientLight) is high (sunny), low (cloudy), or very low (night-time), and finally, (iii) if the user is a novice or experienced user, based on a threshold value of usage time.

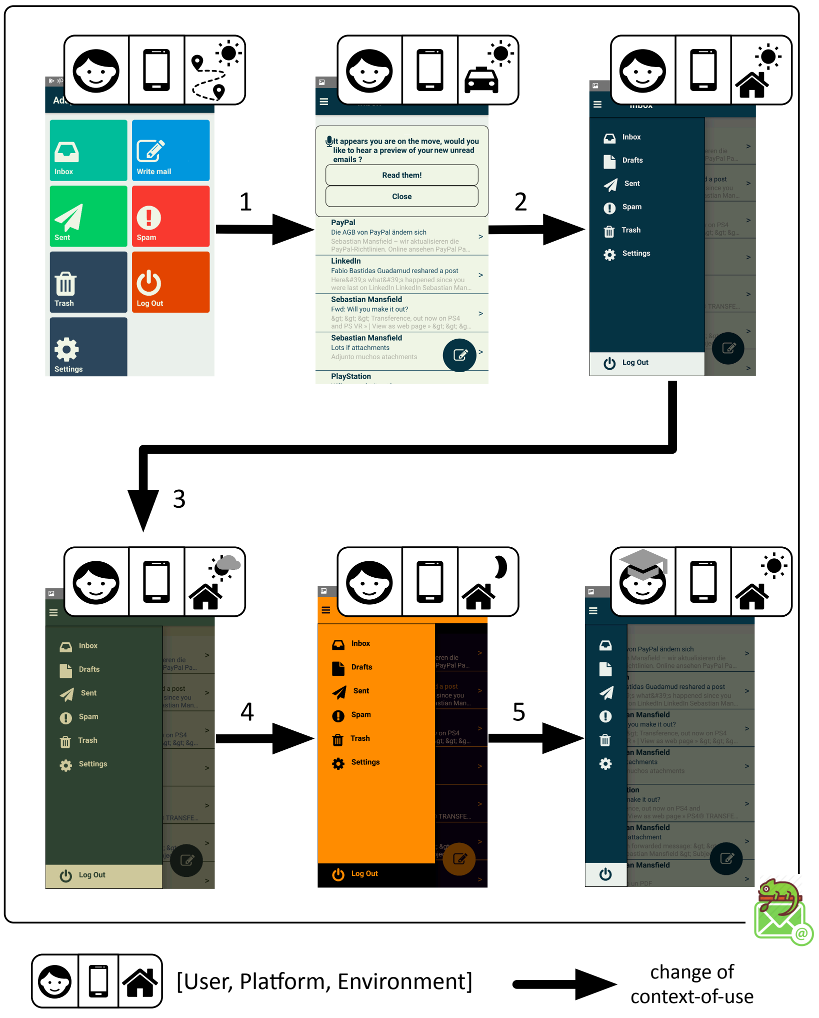


Fig. 7.10 E-mail application: UI adaptations according to different context changes

The first state (left upper corner in Fig. 7.10) represents a novice user on the move and experiencing high brightness levels (ambient light). The corresponding self-adaptive UI recognizes the context properties *movement* and *ambientLight* and uses a grid layout to simplify haptic interaction.

Figure 7.11 shows exactly this change *CoU1* to *CoU2* (compare Figure 7.10) using an object diagram. The depicted context-of-use object model excerpt in Figure 7.11 illustrates how different contextual parameters regarding user, platform, and environment change.

In response to the context change (depicted in Fig. 7.10 as labelled arrows – in this case with Label 1) leading to a state where the user is now in a moving vehicle, the UI switches its modality to audio-based interaction, offering to read new e-mails aloud and enabling control of the application via audio commands. When the user is immobile for some time (and can be assumed to be seated in a building – see Label 2), the UI responds by reverting to standard

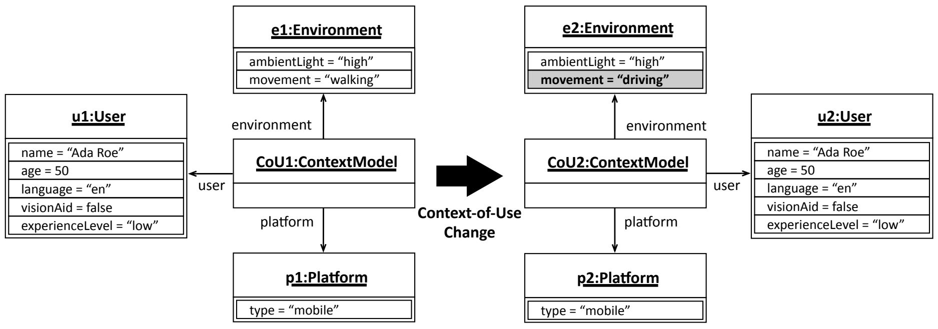
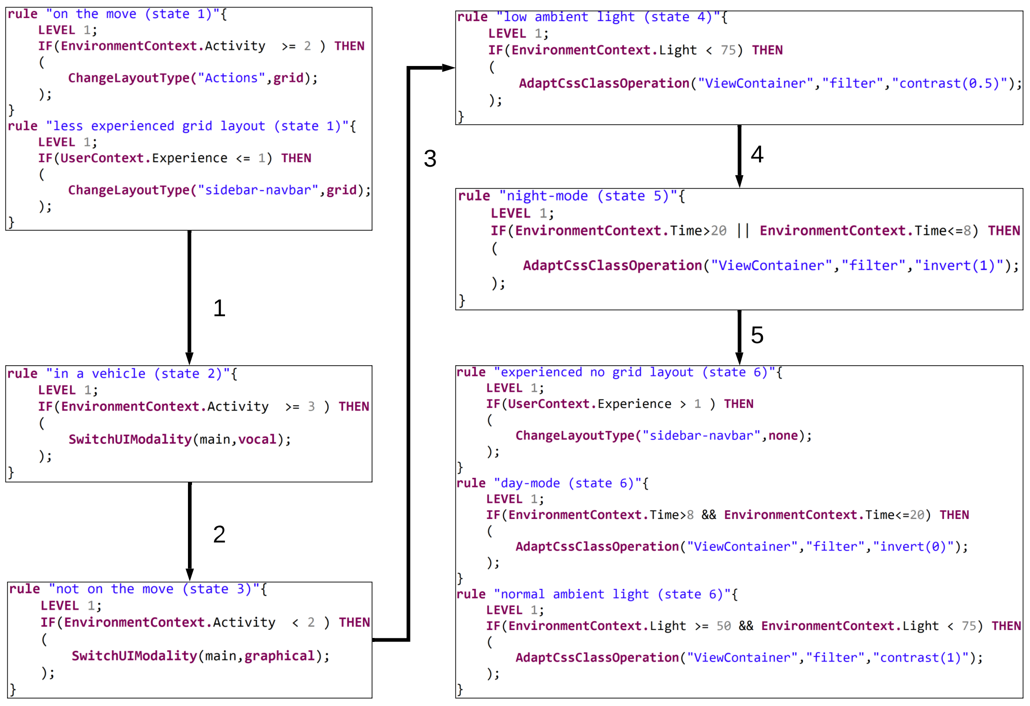


Fig. 7.11 E-mail application: Context-of-use object model excerpts


```
rule "on the move (state 1)"{
    LEVEL 1;
    IF(EnvironmentContext.Activity  >= 2 ) THEN
    (
        ChangeLayoutType("Actions",grid);
    );
}
rule "less experienced grid layout (state 1)"{
    LEVEL 1;
    IF(UserContext.Experience <= 1) THEN
    (
        ChangeLayoutType("sidebar-navbar",grid);
    );
}
```

```
rule "in a vehicle (state 2)"{
    LEVEL 1;
    IF(EnvironmentContext.Activity  >= 3 ) THEN
    (
        SwitchUIModality(main,vocal);
    );
}
```

```
rule "not on the move (state 3)"{
    LEVEL 1;
    IF(EnvironmentContext.Activity  < 2 ) THEN
    (
        SwitchUIModality(main,graphical);
    );
}
```

```
rule "low ambient light (state 4)"{
    LEVEL 1;
    IF(EnvironmentContext.Light < 75) THEN
    (
        AdaptCssClassOperation("ViewContainer","filter","contrast(0.5)");
    );
}
```

```
rule "night-mode (state 5)"{
    LEVEL 1;
    IF(EnvironmentContext.Time>20 || EnvironmentContext.Time<=8) THEN
    (
        AdaptCssClassOperation("ViewContainer","filter","invert(1)");
    );
}
```

```
rule "experienced no grid layout (state 6)"{
    LEVEL 1;
    IF(UserContext.Experience > 1 ) THEN
    (
        ChangeLayoutType("sidebar-navbar",none);
    );
}
rule "day-mode (state 6)"{
    LEVEL 1;
    IF(EnvironmentContext.Time>8 && EnvironmentContext.Time<=20) THEN
    (
        AdaptCssClassOperation("ViewContainer","filter","invert(0)");
    );
}
rule "normal ambient light (state 6)"{
    LEVEL 1;
    IF(EnvironmentContext.Light >= 50 && EnvironmentContext.Light < 75) THEN
    (
        AdaptCssClassOperation("ViewContainer","filter","contrast(1)");
    );
}
```

Fig. 7.12 E-mail application: applied UI adaptation rules for above described scenario

haptic-based modality and additionally uses a list of icons instead of a grid for more efficient screen space usage. The next two context changes (Label 3 and 4), represent changes in brightness level to low brightness and night-time. The UI responds to low brightness levels by dimming the screen and using sepia tones instead of white/black, and to night-time by inverting the color scheme. The final context change (Label 5) is triggered when the user passes a certain usage-time threshold. The UI assumes that the user must now be accustomed enough to the icons and saves screen space by removing the explanatory labels for each icon.

Figure 7.12 shows an overview of the described UI adaptation process where for each state the applied UI adaptation rules are depicted.

As a summary, also the second case study shows a successful application of our model-driven development approach for supporting the development of an e-mail application with context management and UI adaptation capabilities. By using our integrated modeling workbench and the corresponding *SAUI-Generator*, we were able to model and generate self-adaptive UIs for the e-mail application. Thus, we can conclude that our solution approach was helpful for addressing the introduced evaluation questions **EQ1**-**EQ3**.

7.1.3 Case studies: Evaluation Discussion

In the following, we discuss and answer the evaluation questions introduced in the beginning of Section 7.1 based on our experiences made as part of the previously described case studies.

EQ1 Does the solution approach support the integrated modeling of self-adaptive UIs by covering relevant concerns such as core UI aspects, context management, and UI adaptation?

With respect to evaluation question EQ1, our experiences gathered during both case studies have shown that our integrated modeling approach for self-adaptive UIs is well-suited to specify core UI, context management and UI adaptation concerns. Especially the usage of our integrated modeling workbench with its support of different modeling views for UI, context, and adaptation has shown that it eases the modeling of self-adaptive UIs and also supports the maintenance of evolving context and adaptation models.

Nevertheless, a limitation of our modeling approach is the lack of modeling support for application or business logic. As *IFML* is not intended to specify application or business logic (and this was also not within the scope of this thesis), we had to manually implement and add the application logic code for the corresponding case studies. This covers on the one hand application functionality, but also data storage and communication that had to be manually implemented for the case studies. Also, some limitations were given due to the development state of the used open source *IFML* Eclipse editor which does not support specific *IFML* model aspects defined in the *IFML* metamodel.

EQ2 Does the solution enable the generation of code for the *Final UI*, *Context Services*, and *Adaptation Services* to automate the development approach for self-adaptive UIs?

Addressing evaluation question EQ2, the experiences from both cases studies have shown that our model-driven development approach enables the generation of self-adaptive UIs. As illustrated in both case studies, the relevant artifacts of the self-adaptive UI: *Final UI* code, as well as code for *Context* and *Adaptation Services* have been generated in an automated way.

Nevertheless, we have to point out that our transformation approach has been implemented as a proof-of-concept implementation covering a subset of *IFML* model elements for generating the UI based on Angular as target UI framework. Due to the very individual structure of some context sources, the code for controlling and managing sensor sources, like APIs, SDKs or libraries, could not be always automatically generated and thus had to be implemented manually.

EQ3 Does the solution approach enable runtime UI adaptation by integrating the generated artifacts *Final UI* code, *Context*, and *Adaptation Services* in an overall rule-based execution environment?

The resulting self-adaptive UIs in both case studies show the practicability of our approach as in both example scenarios, we were able to generate self-adaptive UIs and to test and use them in action. The identified context changes and the UI adaptations in action demonstrate that the generated self-adaptive UIs are able to continuously monitor their context-of-use parameters and automatically adapt the UI at runtime.

As a limitation of the execution part of our solution, it has to be noticed that it is still a challenging task to correctly time the adaptation operations as time delays in the communication might happen. For example, the camera is used for sensing the mood of the user and it takes time to get the mood information from the face detection API from a different server. Thus, there is potential for improvement to better time the triggered UI adaptations.

7.2 Usability Study

In the previous section, we have presented case studies based on two different application scenarios to evaluate the applicability of our model-driven development approach. This section deals with the usability evaluation of the resulting self-adaptive UIs which have been developed based on our model-driven development approach. For the usability study, we aim to evaluate our solution approach with respect to the following central question:

EQ4 Does the solution approach result in self-adaptive UIs which are accepted by the end-users?

To find an answer for this evaluation question, one can possibly use classical usability evaluation methods like usability tests, interviews or cognitive walkthroughs. However, as already argued in the beginning of this thesis, these methods are not sufficient for a proper evaluation of UI adaptation features (*a posteriori* analysis, no consideration of current context-of-use when adaptations are triggered). Therefore, in the following, we first introduce a novel on-the-fly usability evaluation solution. After that, we describe a usability experiment conducted based on this usability evaluation solution. Finally, we present and discuss the results of this usability study.

7.2.1 On-the fly Usability Evaluation Solution

Our on-the-fly usability testing solution for UI adaptation features targets rule-based UI adaptation approaches, continuously monitors context information about context characteristics, and collects instant user feedback about triggered UI adaptation features. To realize such an on-the-fly usability testing solution, we extended our existing rule-based UI adaptation approach with capabilities to support continuous context monitoring and collecting context-driven instant user feedback.

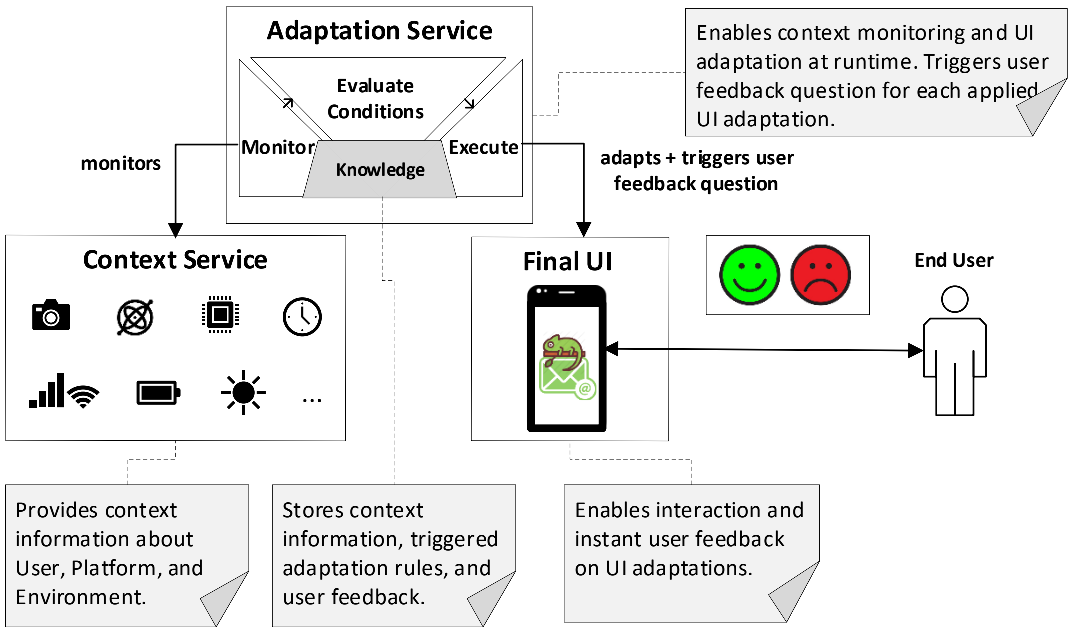

Adaptation Service
Evaluate Conditions
Monitor
Knowledge
Execute
Enables context monitoring and UI adaptation at runtime. Triggers user feedback question for each applied UI adaptation.
monitors
adapts + triggers user feedback question
Context Service
...
Final UI
End User
Provides context information about User, Platform, and Environment.
Stores context information, triggered adaptation rules, and user feedback.
Enables interaction and instant user feedback on UI adaptations.

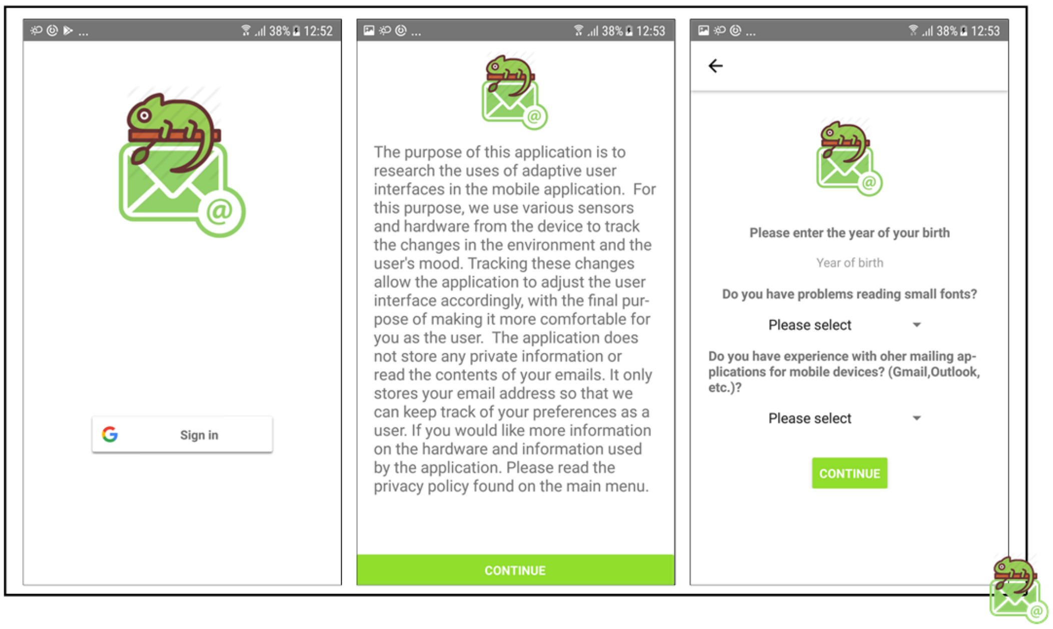
Fig. 7.13 Human-in-the-loop: On-the-fly usability testing of UI adaptation features

Figure 7.13 illustrates the main idea of our on-the-fly usability testing solution for UI adaptations. As presented earlier, IBM's MAPE-K loop [IBM05] was used as an architectural pattern to realize the UI adaptation process. In our realization, we have derived the components *Final UI*, *Context Service*, and *Adaptation Service* which are deployed and running on the same target platform. The *Adaptation Service* is responsible for monitoring the *Context Service* and adapting the *Final UI*, which is the user interface of a target platform. For this purpose, the *Adaptation Service* monitors various context information gathered through integrated sensors and cameras based on the *Context Service*. The context information is then analyzed by evaluating whether UI adaptation rules match to the current context-of-use situation. If this is the case, the execute component is responsible for executing the UI adaptation operations on the *Final UI*.

While the previously described components are essential for supporting rule-based UI adaptation in general, further components and mechanisms are needed to support on-the-fly usability testing of UI adaptation features. To this end, we integrated an instant user feedback mechanism into the context monitoring and UI adaptation loop. The feedback mechanism allows users to explicitly rate the triggered UI adaptations. To realize this feedback mechanism, our *Adaptation Service* triggers simultaneously to each UI adaptation operation a feedback question for the end user. This way, end users can give positive or negative feedback about the applied UI adaptations, but they can also ignore the feedback mechanism and concentrate on their main application task. As Figure 7.13 shows, there is a *Knowledge* base which is responsible for storing all context information before and after a context change occurred (that lead to a UI adaptation triggering), all triggered adaptation rules, and the corresponding instant user feedback. Based on the stored information, it is possible to analyze the acceptance of UI adaptations based on the current context of the user and the user's feedback. As the *Knowledge* base characterizes a key component and asset for the data-driven usability evaluation, all context information were collected in a central database.

Implementation

We implemented and applied our on-the-fly usability testing solution for UI adaptation features based on the e-mail application which was presented in Section 7.1.2. We decided to use an e-mail application as e-mail clients are one of the most frequently used applications on different target platforms. People read and write mails while commuting to work, before going to sleep, while walking, watching TV or doing different other activities. The implemented e-mail application is similar to Gmail and provides full e-mail services like send, receive, forward, delete, and compose mails. In addition, a file manager is provided allowing users to download attachments and to open them using different applications on their target platform. Figure 7.14 shows the starting screens of the e-mail application where login, participation info, and a preliminary questionnaire are depicted.



The purpose of this application is to research the uses of adaptive user interfaces in the mobile application. For this purpose, we use various sensors and hardware from the device to track the changes in the environment and the user's mood. Tracking these changes allow the application to adjust the user interface accordingly, with the final purpose of making it more comfortable for you as the user. The application does not store any private information or read the contents of your emails. It only stores your email address so that we can keep track of your preferences as a user. If you would like more information on the hardware and information used by the application. Please read the privacy policy found on the main menu.

CONTINUE

Please enter the year of your birth

Year of birth

Do you have problems reading small fonts?

Please select

Do you have experience with oher mailing applications for mobile devices? (Gmail,Outlook, etc.)?

Please select

CONTINUE

Fig. 7.14 Login (left), participation info (middle), and preliminary questionnaire (right) screens

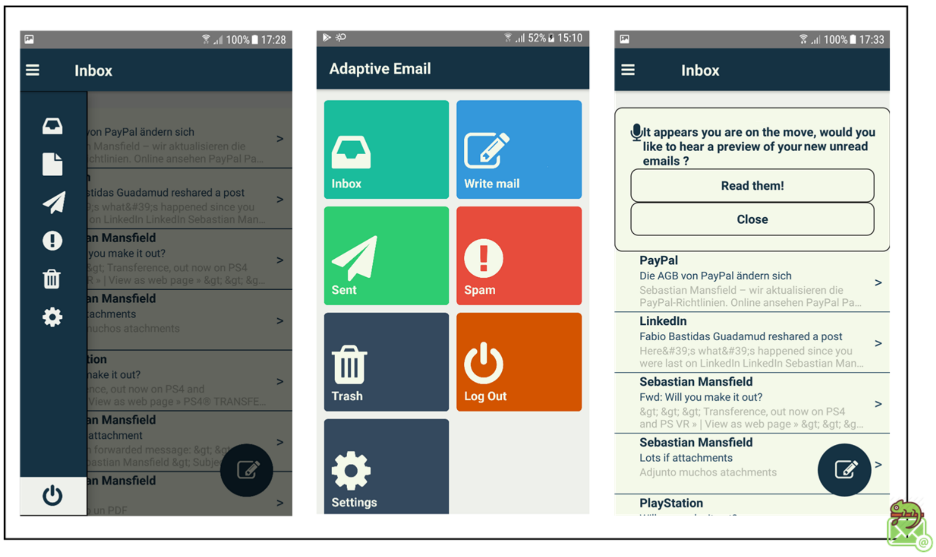
Beside the core mail application logic, which was manually implemented, we used our model-driven development approach to devise a self-adaptive UI for the e-mail application as explained in the second case study in Section 7.1.2). Inspired by [Pat13], we derived and implemented 28 UI adaptation features based on *AdaptML* that cover different adaptation techniques. like task change operation, navigation change operation, layout change operation or modality change operation.

Before giving an overview and detailed description of the implemented UI adaptation features for the e-mail application, we firstly describe the main UI adaptation categories that were used for the e-mail application. In general, the following main adaptation categories were used for the self-adaptive UI of the e-mail application.

TaskChange ***(T)*** adaptations support UI adaptations by flexibly showing and hiding interaction elements of the UI. *LayoutChange* ***(L)*** adaptations enable UI adaptations that change the appearance of the UI by modifying presentation aspects like color, font (size), position, etc. *ModalityChange* ***(M)*** adaptations relate to changes where a switch from one interaction mode to another one is made (e.g., from the graphical to the vocal UI). While these UI adaptation categories are derived from *AdaptML* directly and characterize a superset of specific UI adaptation operations, further useful adaptation categories were implemented for the usability study of the e-mail application. With this regard, we noticed that deactivation rules are helpful to deactivate specific UI adaptation rules if the UI adaptation is not needed any longer. Therefore, we introduced and implemented a further type of UI adaptation category which is called *Deactivation **(D)***. Furthermore, we have identified a category for UI adaptations, which are highly related to a specific domain and application area. For the e-mail application of the usability study, for example, automatic download of attachments based on the internet connection speed can be see as such a concern. We call this type of UI adaptations *Functional **(F)*** adaptations as they provide an additional value for the underlying application going beyond the previously mentioned categories. Finally, it is important to note that adaptation features from the previous categories can be combined to a more complex adaptation feature, so called *Combined **(C)*** adaptation. In the following, we describe the implemented adaptation features and map them to the corresponding adaptation categories.

- Rule 1: Sad Mood. When the user is in a sad mood (detected through the face camera), a feedback text field is shown where the user can enter textual feedback. **(*T*)**
- Rule 2: Good Mood. When the user is in a good mood (detected through the face camera), a motivational quotation is shown. **(*T*)**
- Rule 3: Neutral Mood. When the user is in a neutral mood (detected through the face camera), Rule 1 and 2 are deactivated and no pop-up menu appears. **(*D*)**
- Rule 4: Iconic UI. If the user has been using the app for a while (usage time), an iconic UI without text labels is shown (see screenshot on the left of Figure 7.15). **(*T*)**
- Rule 5: Minimized Navigation. If the user has been using the app for a while (usage time), extra information and text labels in the navigation bar are hidden. **(*T*)**
- Rule 6: Detailed UI. If the user is not experienced (based on preliminary questionnaire), additional information in the the UI and navigation are displayed. **(*T*)**
- Rule 7: Low Ambient Light. If the environmental lighting condition is low, adapt the UI color scheme and contrast to improve better readability in the dark. (*Layout*)
- Rule 8: Normal Ambient Light. If the environmental lighting condition is normal, adapt the UI color scheme and contrast to default mode. (*Layout*)
- Rule 9: High Ambient Light. If the environmental lighting condition is high, increase the UI contrast to improve readability. **(*L*)**
- Rule 10: Day Mode. When the day starts (clock timer), all UI elements are shown in the layout of the day mode. **(*L*)**

- Rule 11: Night Mode. When the day ends (clock timer), all UI elements are shown in the layout of the night mode. ***(L)***
- Rule 12: Vocal UI. When the user is moving (accelerometer), all unread e-mails are read out using a text-to-speech (TTS) service (see screenshot on the right of Figure 7.15). ***(M)***
- Rule 13: Impaired Vision. If the user has vision problems (preliminary questionnaire), the font size of the UI elements is increased. ***(L)***
- Rule 14: Middle Aged User. Based on the entered age of the user (preliminary questionnaire), all UI elements are displayed in the default layout with a slightly larger font size. ***(L)***
- Rule 15: Older User. Based on the entered age of the user (preliminary questionnaire), all UI elements are displayed in the grid layout with a bigger font size (see screenshot in the middle of Figure 7.15). ***(L)***
- Rule 16: Younger User. Based on the entered age of the user (preliminary questionnaire), all UI elements are displayed in the default layout with the default font size. ***(L)***
- Rule 17: Novice User. Based on the entered experience level of the user (preliminary questionnaire), the UI is adapted to the grid layout. ***(L)***
- Rule 18: Expert User. Based on the entered experience level of the user (preliminary questionnaire), the UI is shown in the default layout. ***(L)***
- Rule 19: Smartphone Mode. If the used device is a smartphone, a combined adaptation is executed where navigation and layout is optimized for smartphone devices. ***(C)***
- Rule 20: Tablet Mode. If the used device is a tablet, a combined adaptation is executed where navigation and layout is optimized for tablet devices. ***(C)***
- Rule 21: Driving Mode. When the user is driving (accelerometer), a combined adaptation is executed where the UI is displayed using the grid layout and additionally the text-to-speech service is activated for reading out mails. ***(C)***
- Rule 22: Driving Mode Off. When the user is still (stopped or reached destination), the Rule 21 is deactivated. ***(D)***



It appears you are on the move, would you like to hear a preview of your new unread emails ?

PayPal
Die AGB von PayPal ändern sich
Sebastian Mansfield – wir aktualisieren die PayPal-Richtlinien. Online ansehen PayPal Pa...

LinkedIn
Fabio Bastidas Guadamud reshared a post
Here&#39;s what&#39;s happened since you were last on LinkedIn LinkedIn Sebastian Man...

Sebastian Mansfield
Fwd: Will you make it out?
&gt; &gt; &gt; Transference, out now on PS4 and PS VR » | View as web page » &gt; &gt; &g...

Sebastian Mansfield
Lots if attachments
Adjunto muchos atachments

PlayStation

Fig. 7.15 UI Adaptations: "iconic" UI (left), "grid" UI (middle), vocal UI (right)

- Rule 23: Low Battery. When the battery level of a mobile platform is low, the UI is displayed in a black and white color scheme to save energy. ***(L)***
- Rule 24: Normal Battery. When the battery level of the mobile platform is normal, rule 23 is deactivated. ***(D)***
- Rule 25: Wi-Fi Connection. If the used device has a Wi-Fi connection, the e-mails are displayed in HTML format (if available) and attachments are automatically downloaded. ***(F)***
- Rule 26: 3/4G Connection. If the used device has a 3/4G connection, the e-mails are displayed in HTML format (if available), but attachments are downloaded only on demand. ***(F)***
- Rule 27: Slow Network. If the used device has a slow connection, the e-mails are displayed in text format and attachments are not automatically downloaded. ***(F)***
- Rule 28: Vocal UI Off. When the user is still (accelerometer), Rule 12 is deactivated. ***(D)***

To realize and execute the described UI adaptation features for the e-mail application, we used our execution environment based on *Nools* (see Section 6.2). The *Nools* rule engine is fed with context information from various integrated sensors and cameras of the target platform. For example, we use different sensors like ambient light, accelerometer, gyroscope, battery or network sensor. Furthermore, we use timer, device information or existing recognition

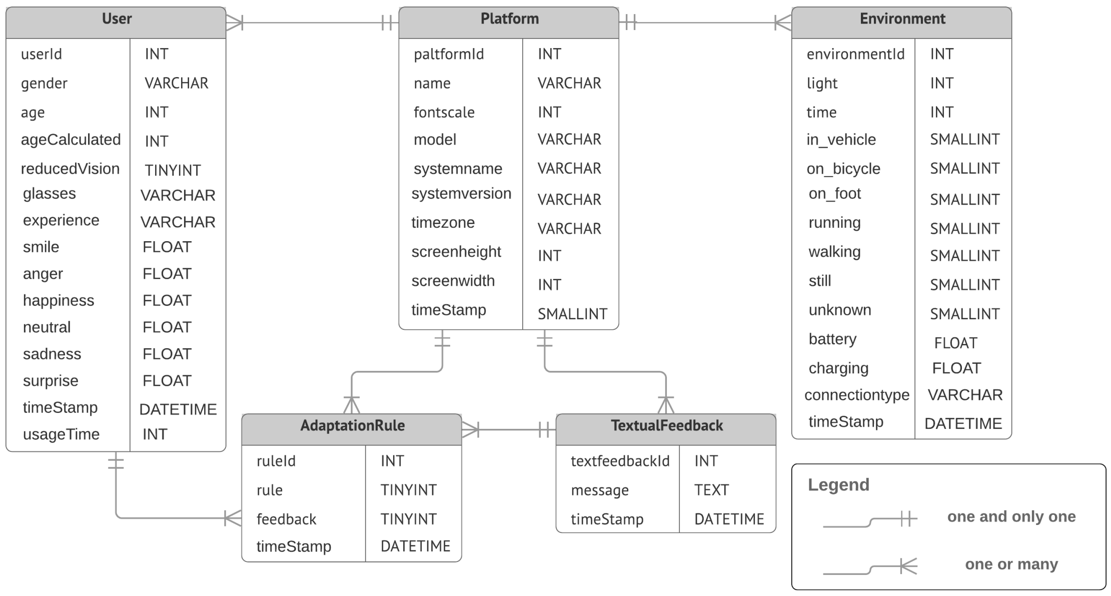


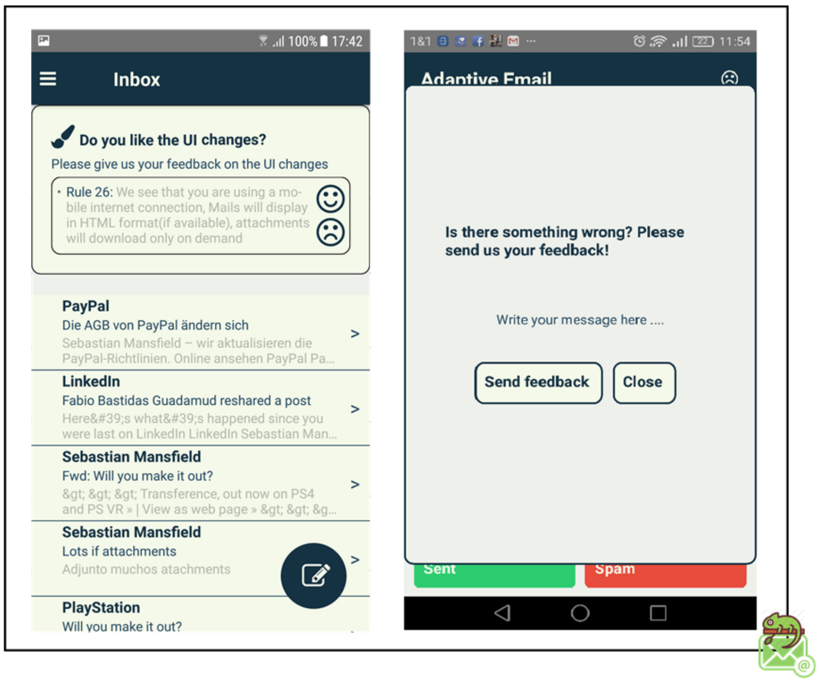
Fig. 7.16 Database schema: Stored context information as extended entity relationship (EER) diagram

services such as *Amazon Rekognition* [1] to derive the user's emotion as well as estimated age or gender based on camera photos. A complete overview about the stored context information in the form of an extended entity-relationship model is depicted in Figure 7.16.

In order to avoid a cold start of the adaptation process, we have integrated a preliminary questionnaire form into our e-mail application (see Figure 7.14) that asks the users for their birth date, ability to read small fonts, and experience with other mailing applications for mobile devices. Based on this starting context information, the UI of the mobile e-mail application can already adapt itself to a suitable starting configuration for the current user.

For realization of the feedback mechanism, we integrated a feedback prompt into the adaptive e-mail application. The left application screen in Figure 7.17 shows how the feedback prompt was placed in the mail application. On the top of the screen the feedback prompt is shown whenever context changes were detected that lead to UI adaptations. The triggered UI adaptations are explained in the feedback prompt (reason for adaptation and UI adaptation rule) and the user is able to provide feedback by clicking the positive or negative smiley indicating whether the user liked the UI adaptation or not. In some cases, for example, when the user is in a bad mood and the application detects this via camera, a feedback prompt in the form of a text field appears (see right application screen in Figure 7.17) that allows the users to provide more detailed feedback. A complete overview of the triggered user

[1] https://aws.amazon.com/de/rekognition/



Do you like the UI changes?

Please give us your feedback on the UI changes

• Rule 26: We see that you are using a mobile internet connection, Mails will display in HTML format(if available), attachments will download only on demand

PayPal
Die AGB von PayPal ändern sich
Sebastian Mansfield – wir aktualisieren die PayPal-Richtlinien. Online ansehen PayPal Pa...

LinkedIn
Fabio Bastidas Guadamud reshared a post
Here&#39;s what&#39;s happened since you were last on LinkedIn LinkedIn Sebastian Man...

Sebastian Mansfield
Fwd: Will you make it out?
&gt; &gt; &gt; Transference, out now on PS4 and PS VR » | View as web page » &gt; &gt; &g...

Sebastian Mansfield
Lots if attachments
Adjunto muchos atachments

PlayStation
Will you make it out?

Is there something wrong? Please send us your feedback!

Write your message here ....

Fig. 7.17 Feedback prompt

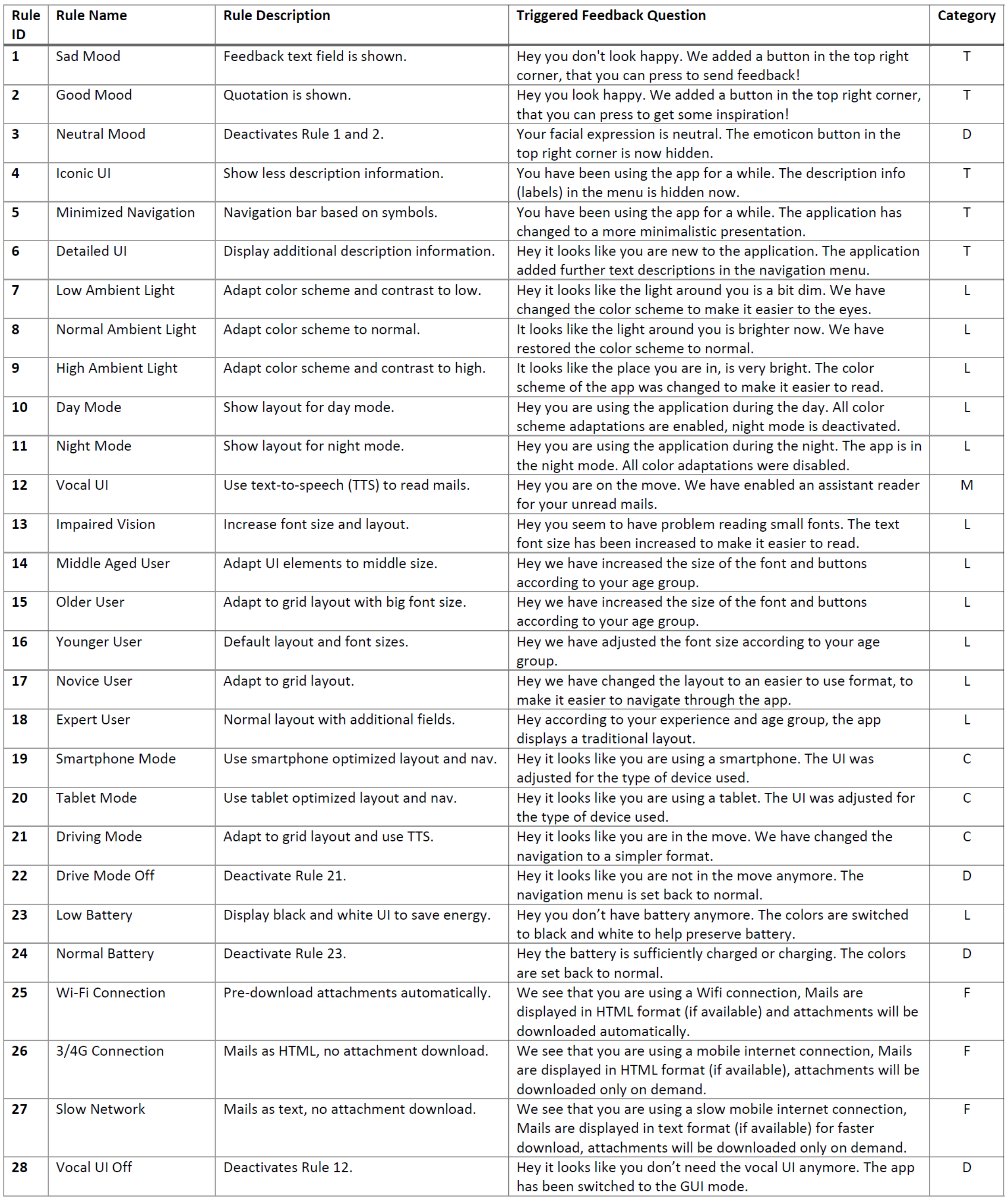
feedback questions can be seen in Figure 7.18 in relation to the corresponding UI adaptation rule whose end-user acceptance is tested. The user feedback questions are string elements which can be freely edited by the developers at design-time to prepare the usability evaluation questions. As already described before, the user feedback questions are then simultaneously triggered with the UI adaptations to gather fresh and instant end-user feedback about the UI adaptations for the current context-of-use.

7.2.2 Usability Experiment and Results

In the following, we describe how the previously presented on-the-fly usability evaluation solution was used to conduct a usability experiment based on the e-mail application. After that, we present the main results and the interpretation of our data-driven usability evaluation.

Usability Experiment

During the usability experiment, the developed e-mail application with context management and UI adaptation capabilities was used for one week by 23 participants, who were recruited via an invitation e-mail. The test users were encouraged to use the application as often and in as many different contexts as possible. All users were made aware of the fact that their interaction with the application would be closely monitored (e.g., using facial recognition). In



Rule ID	Rule Name	Rule Description	Triggered Feedback Question	Category
1	Sad Mood	Feedback text field is shown.	Hey you don't look happy. We added a button in the top right corner, that you can press to send feedback!	T
2	Good Mood	Quotation is shown.	Hey you look happy. We added a button in the top right corner, that you can press to get some inspiration!	T
3	Neutral Mood	Deactivates Rule 1 and 2.	Your facial expression is neutral. The emoticon button in the top right corner is now hidden.	D
4	Iconic UI	Show less description information.	You have been using the app for a while. The description info (labels) in the menu is hidden now.	T
5	Minimized Navigation	Navigation bar based on symbols.	You have been using the app for a while. The application has changed to a more minimalistic presentation.	T
6	Detailed UI	Display additional description information.	Hey it looks like you are new to the application. The application added further text descriptions in the navigation menu.	T
7	Low Ambient Light	Adapt color scheme and contrast to low.	Hey it looks like the light around you is a bit dim. We have changed the color scheme to make it easier to the eyes.	L
8	Normal Ambient Light	Adapt color scheme to normal.	It looks like the light around you is brighter now. We have restored the color scheme to normal.	L
9	High Ambient Light	Adapt color scheme and contrast to high.	It looks like the place you are in, is very bright. The color scheme of the app was changed to make it easier to read.	L
10	Day Mode	Show layout for day mode.	Hey you are using the application during the day. All color scheme adaptations are enabled, night mode is deactivated.	L
11	Night Mode	Show layout for night mode.	Hey you are using the application during the night. The app is in the night mode. All color adaptations were disabled.	L
12	Vocal UI	Use text-to-speech (TTS) to read mails.	Hey you are on the move. We have enabled an assistant reader for your unread mails.	M
13	Impaired Vision	Increase font size and layout.	Hey you seem to have problem reading small fonts. The text font size has been increased to make it easier to read.	L
14	Middle Aged User	Adapt UI elements to middle size.	Hey we have increased the size of the font and buttons according to your age group.	L
15	Older User	Adapt to grid layout with big font size.	Hey we have increased the size of the font and buttons according to your age group.	L
16	Younger User	Default layout and font sizes.	Hey we have adjusted the font size according to your age group.	L
17	Novice User	Adapt to grid layout.	Hey we have changed the layout to an easier to use format, to make it easier to navigate through the app.	L
18	Expert User	Normal layout with additional fields.	Hey according to your experience and age group, the app displays a traditional layout.	L
19	Smartphone Mode	Use smartphone optimized layout and nav.	Hey it looks like you are using a smartphone. The UI was adjusted for the type of device used.	C
20	Tablet Mode	Use tablet optimized layout and nav.	Hey it looks like you are using a tablet. The UI was adjusted for the type of device used.	C
21	Driving Mode	Adapt to grid layout and use TTS.	Hey it looks like you are in the move. We have changed the navigation to a simpler format.	C
22	Drive Mode Off	Deactivate Rule 21.	Hey it looks like you are not in the move anymore. The navigation menu is set back to normal.	D
23	Low Battery	Display black and white UI to save energy.	Hey you don’t have battery anymore. The colors are switched to black and white to help preserve battery.	L
24	Normal Battery	Deactivate Rule 23.	Hey the battery is sufficiently charged or charging. The colors are set back to normal.	D
25	Wi-Fi Connection	Pre-download attachments automatically.	We see that you are using a Wifi connection, Mails are displayed in HTML format (if available) and attachments will be downloaded automatically.	F
26	3/4G Connection	Mails as HTML, no attachment download.	We see that you are using a mobile internet connection, Mails are displayed in HTML format (if available), attachments will be downloaded only on demand.	F
27	Slow Network	Mails as text, no attachment download.	We see that you are using a slow mobile internet connection, Mails are displayed in text format (if available) for faster download, attachments will be downloaded only on demand.	F
28	Vocal UI Off	Deactivates Rule 12.	Hey it looks like you don’t need the vocal UI anymore. The app has been switched to the GUI mode.	D

Fig. 7.18 Overview of feedback question for each UI adaptation

the following, we present some facts and figures about the test user group and the encountered contexts.

The participant sample consisted of 10 male and 13 female users. This information was inferred from the facial recognition component. The facial recognition component also detected that 8 of the 23 users used reading glasses during the experiment.

Based on the prompt asking users for their date of birth when starting the application for the first time, the test users were between 22 and 55 years old. The majority of the users were in their twenties and thirties. From the 23 users, only 2 were 40 years or older. Eight of the users were between 30 and 40 years old. The rest of the users were younger than 30 years. Three users stated that they have problems reading small fonts while 20 answered that they do not have problems with small fonts. When asked for their experience with other e-mailing applications for mobile devices 13 users rated their experience level as high, 5 as medium, 3 as low and 2 of them stated that they have no experience.

Usability Evaluation Results

The data that was collected during the usability experiment allows us to evaluate the usability of the UI adaptation features in detail. Our main goal is to examine if the UI adaptations are accepted by the end-users (EQ4). Furthermore, we aim to understand how the user context influences the acceptance of the different UI adaptations. To investigate both aspects, an essential requirement is that the application was indeed used in different contexts by the end-users. To make sure that this is the case, we first conduct a preliminary analysis of the collected data, where the total usage time, the amount of given user feedback, and different context-of-use situations while feedback was given are investigated.

Preliminary Analysis:

The goal of this preliminary analysis is to confirm that the participants did use the mobile application in different contexts. Figure 7.19 shows the end-users and their aggregated usage time of the application. On average, each end-user spent almost 15 minutes in the application during the usability experiment. The user who used the application the most spent a total of 42 minutes in the application while the user who used the application the least only spent about two minutes in total in the application.

In Figure 7.20 it is shown how often the users gave feedback during each hour of the day. The users gave the most feedback in the time interval from 7 am to 8 pm. However, we also received responses to UI adaptations during the rest of the day. This usage pattern provides several different context-of-use situations for our analysis.

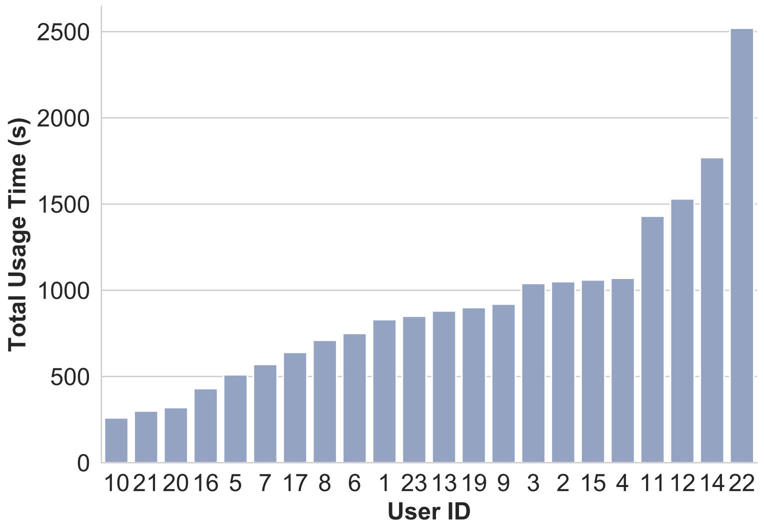


Fig. 7.19 Total usage time of the application for each user

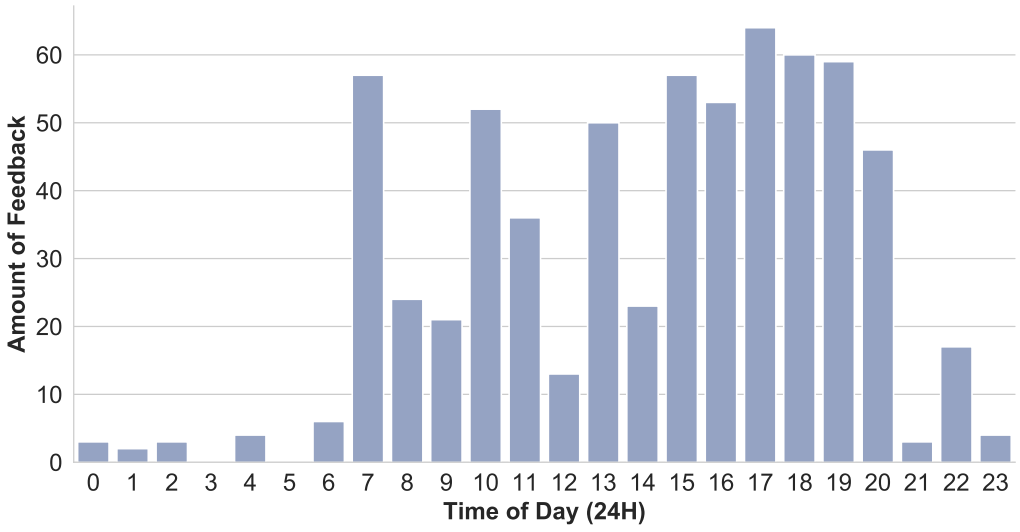


Fig. 7.20 Amount of given user feedback for each hour of the day

To assess the context-of-use based on the used platform sensors, the application tries to predict the environmental circumstances (still, walking, in vehicle, etc.) of the user. In Figure 7.21 the amount of feedback that was given during different environmental conditions regarding movement is shown.

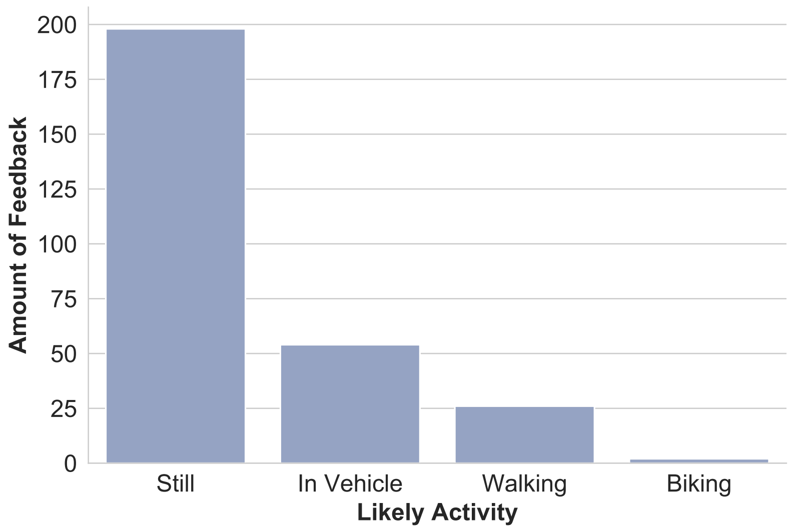


Fig. 7.21 Amount of user feedback collected for each predicted user activity. Only responses are shown for which the activity could be predicted with high certainty (280 cases).

We can infer that the application was used while commuting or in a fixed environment. All these context changes were used by the application to adapt the user interface. Furthermore, the Figure 7.21 underlines the fact that user feedback about UI adaptations was received under dynamically changing context-of-use situations.

Revisiting Evaluation Question EQ4:

In Table 7.1, we can see some data about the context changes. There were 104404 detected context changes from all devices in the experiment. Of these, only 37465 triggered an adaptation by the rule engine. However, users gave feedback on the adaptation rules in only 663 cases. Every time an adaptation rule received feedback, the previous context additionally to the current context was saved.

Context changes		
Detected	Significant	Feedback
104404	37465	663

Table 7.1 Context change information

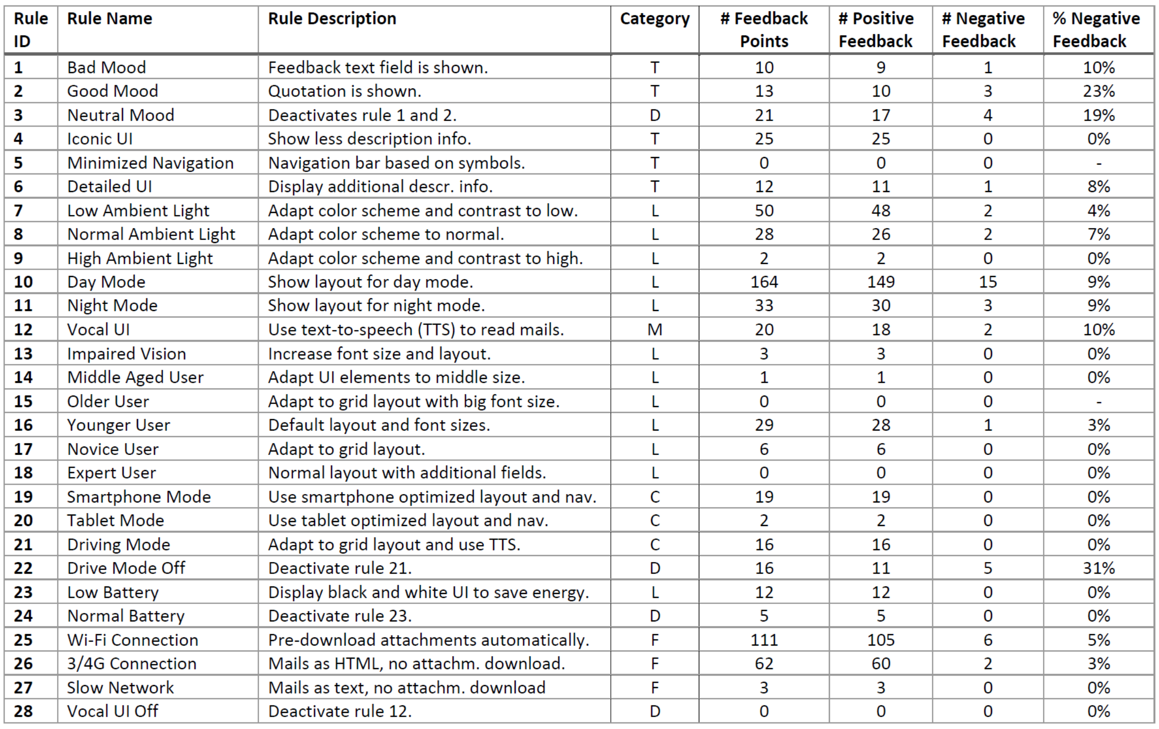
User feedback statistics			
Total	Positive	Negative	% Positive
663	616	47	93

Table 7.2 User feedback

For an initial evaluation of the research question regarding usability, an analysis of the collected user feedback is done. In table 7.2 some descriptive statistics about the feedback provided by the users through the application are shown. In total, the users gave positive feedback in 616 cases and negative feedback in 47 cases. With about 93% of the feedback provided by users being positive this means that most of the user interface adaptations were liked by the users. As a conclusion of this, the evaluation question EQ4 can be answered positively, meaning that the UI adaptations were accepted by the participants of the usability study.

Direct feedback:
To complement the analysis, users were asked to give a short optional written feedback about the application and the adaptation rules. Although only seven users provided text feedback, their contributions can give us a more in-depth understanding of their feedback responses in the application and their perception of the usability of the application. Most of the users reported that they liked the application and its adaptations, and found them useful. Two users reported that although they liked the ability of the application to change colors in different context situations, the color schemes chosen by the adaptations were not to their liking. Two users reported that rule 22 deactivated the driving mode at traffic lights or other temporary stops. This was found problematic by the users, hence the negative feedback.

Further Analysis of Usability Evaluation Data:
To gain more insights regarding the received feedback for the different rules, we created an overview of the UI adaptation features with the amount of received positive and negative feedback shown in Figure 7.22. We can see on the far left, the *Rule ID* of an adaptation feature, followed by the *Rule Name*, a short *Rule Description*, the *Category* (*Task-Feature (T)*, *Layout (L)*, *Modality (M), Functional (F), Combined (C), Deactivation (D)*) where the UI adaptation feature belongs to, the total number of feedback points, the number of positive feedback responses, negative feedback responses, and finally the percentage of negative feedback. Some of the UI adaptation features received no feedback at all, some received only positive feedback and some of them a combination of both positive and negative feedback. Four rules (14%) received no feedback from any user. This could be because the conditions for those rules were not met or users simply decided not to give feedback on these rules



Rule ID	Rule Name	Rule Description	Category	# Feedback Points	# Positive Feedback	# Negative Feedback	% Negative Feedback
1	Bad Mood	Feedback text field is shown.	T	10	9	1	10%
2	Good Mood	Quotation is shown.	T	13	10	3	23%
3	Neutral Mood	Deactivates rule 1 and 2.	D	21	17	4	19%
4	Iconic UI	Show less description info.	T	25	25	0	0%
5	Minimized Navigation	Navigation bar based on symbols.	T	0	0	0	-
6	Detailed UI	Display additional descr. info.	T	12	11	1	8%
7	Low Ambient Light	Adapt color scheme and contrast to low.	L	50	48	2	4%
8	Normal Ambient Light	Adapt color scheme to normal.	L	28	26	2	7%
9	High Ambient Light	Adapt color scheme and contrast to high.	L	2	2	0	0%
10	Day Mode	Show layout for day mode.	L	164	149	15	9%
11	Night Mode	Show layout for night mode.	L	33	30	3	9%
12	Vocal UI	Use text-to-speech (TTS) to read mails.	M	20	18	2	10%
13	Impaired Vision	Increase font size and layout.	L	3	3	0	0%
14	Middle Aged User	Adapt UI elements to middle size.	L	1	1	0	0%
15	Older User	Adapt to grid layout with big font size.	L	0	0	0	-
16	Younger User	Default layout and font sizes.	L	29	28	1	3%
17	Novice User	Adapt to grid layout.	L	6	6	0	0%
18	Expert User	Normal layout with additional fields.	L	0	0	0	0%
19	Smartphone Mode	Use smartphone optimized layout and nav.	C	19	19	0	0%
20	Tablet Mode	Use tablet optimized layout and nav.	C	2	2	0	0%
21	Driving Mode	Adapt to grid layout and use TTS.	C	16	16	0	0%
22	Drive Mode Off	Deactivate rule 21.	D	16	11	5	31%
23	Low Battery	Display black and white UI to save energy.	L	12	12	0	0%
24	Normal Battery	Deactivate rule 23.	D	5	5	0	0%
25	Wi-Fi Connection	Pre-download attachments automatically.	F	111	105	6	5%
26	3/4G Connection	Mails as HTML, no attachm. download.	F	62	60	2	3%
27	Slow Network	Mails as text, no attachm. download	F	3	3	0	0%
28	Vocal UI Off	Deactivate rule 12.	D	0	0	0	0%

Fig. 7.22 Overview: UI adaptation features and received user feedback

(users could ignore the feedback prompt and continue using the application). Eleven rules (almost 40%) received only positive feedback. No rules received only negative feedback.

From the 13 rules that received negative feedback, ten got mostly positive feedback with only 10% or less of the received feedback being negative. The rules that got most negative feedback are rule 22 (*Drive Mode Off*) with 31% negative feedback, rule 2 (*Good Mood*) with 23% negative feedback, and rule 3 (*Neutral Mood*) with 19% negative feedback. Two of these rules, namely 3 and 22 are deactivation rules, meaning that the purpose of this adaptation rules is to deactivate the effects of rules whose condition is no longer active. Please note that looking only at the percentage of negative feedback can be in some cases a little misleading, as some rules received very little feedback overall. In these cases, a single negative response can have a significant impact on the overall percentage value.

Relying on the shown results in Figure 7.22, by separately calculating the average amount of positive and negative feedback for each adaptation category, we derived a categorization of UI adaptation techniques according to their degree of end-user satisfaction. The results of this categorization are shown in Figure 7.23.

As the amount of feedback data is low, especially regarding the number of negative feedback, and the received amount of feedback is not balanced for each category (see table column *Category* in Figure 7.22), the results should be cautiously interpreted. However, we

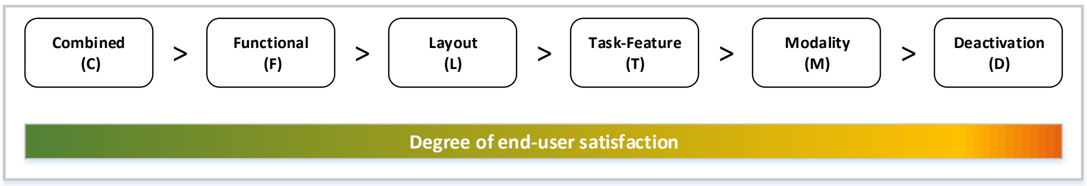


Fig. 7.23 Categorization of UI adaptation techniques according to their degree of end-user satisfaction

can observe that *Combined (C)* UI adaptation features that were aggregated based on basic UI adaptation features such as rule 19 (a combination of layout and navigation adaptation) have the largest impact on increasing user satisfaction. The second category which has also a big impact on increasing user satisfaction is *Functional (F)* UI adaptation. *Functional* UI adaptations, such as rule 25 are domain specific changes that provide additional value to the application for example by downloading the attachments if a high network connection is available. Good results regarding user satisfaction can be also reached by *Layout (L)* and *Task-Feature (T)* set adaptation rules. However, more attention has to be paid when it comes to the design of adaptive UIs incorporating *Modality (M)* changes and *Deactivation* rules. UI adaptations regarding *Modality* changes have potential for improving user satisfaction, but as the textual feedback by the users shows, many switches between two different modalities can also confuse the users and result in negative feedback. We observed that *Deactivation* rules for UI adaptations can have a negative influence on user satisfaction if timing and mapping to the current context-of-use is not well suited. The textual feedback by the participants shows, for example, that deactivation rule 22 was problematic when sudden context changes from the state driving to still were recognized and the vocal UI was deactivated. In this case, a warning message or better timing of UI adaptations was desired by the users.

To gain a better understanding of the provided feedback, we also performed a graphical exploratory data analysis. We treated the feedback of the user to a UI adaptation as the dependent variable. The independent variables were composed of variables describing the context-of-use when the UI adaption was triggered and the context-of-use when the user feedback was given. The variables describing the context-of-use consist of the user, platform and environment variables. We graphically analyzed the interaction between the dependent and the independent variables with the goal to find patterns and anomalies. As it is not possible to find any interesting interaction effects in case that (almost) all considered data points represent positive feedback, we only considered rules for which at least 4 negative responses exist. We note that due to the very limited amount of negative responses for all rules, it is not possible to draw any statistical significant conclusions based on the given data.

However, we show based on the following example that our analysis allowed us to identify an unintended behavior of the application based on the collected data.

Example: Analysis for Adaptation Rule 10
Rule number 10 is activating the day mode by changing the layout color scheme. During the experiment, users gave feedback on this rule 164 times. From these responses, 149 were positive while only 15 were negative. During the exploratory data analysis we examined the relationship between the user feedback for this rule and the context variable describing the time of day using the visualization shown by Figure 7.24. It shows the distribution of positive and negative feedback responses via a density plot (created using kernel density estimation). We can see that while negative feedback was given at all times of the day, most of the negative feedback was received after 15:30. After a closer investigation, we derived the following explanation for this observation: Rule 10 correctly activates the day mode and displays that change to the user in the form of a feedback prompt. However, in cases where the application is used for the first time of the day in the evening, this might confuse users as they expect the activation and not the deactivating of the night mode (which is implicitly done when activating the day mode). In fact, asking the user for feedback in these cases was not an intended behavior of the application, because the user should have been only asked for feedback on recent context changes when starting the application. Our data-driven approach allowed us to identify this confusing and unintended behavior of this adaptation rule.

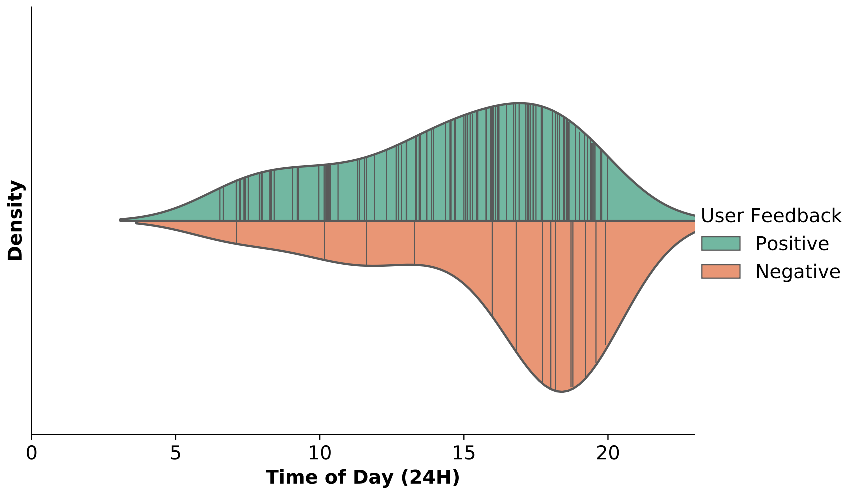


Fig. 7.24 Distribution of positive and negative feedback responses over the day. The green (top) and red (bottom) colored areas show the density estimates of the time of day for positive and negative responses, respectively. The vertical lines represent the underlying datapoints.

7.2.3 Usability Study: Evaluation Discussion

In the following, we discuss the main results of the conducted usability study by summarizing the lessons learned and describing the limitations of the study.

Lessons Learned

From user's perspective our data-driven usability evaluation study shows that self-adaptive UIs have potential to improve usability of the UI. Especially the amount of positive feedback which was collected during the experimental phase shows that self-adaptive UIs enable an improved interaction with UIs which highly increase end-user satisfaction.

Based on the conducted usability experiment and analysis of the results, we derived lessons learned about specific UI adaptation categories as well as about the relationship between context-of-use and triggering UI adaptations.

Based on the conducted usability experiment, we identified that users should only be asked for feedback on adaptions that concern context changes that the user is still aware of. UI adaptations regarding an outdated context change (which happened in the past) should just be discarded or applied without asking or informing the user, who would not be able to understand it anyway as the context-of-use has probably already completely changed. This is especially the case when starting the application after a while. In this case, the user no longer knows the context of the last usage of the application.

Furthermore, we identified that deactivation rules should have a (configurable) time window before they fire. This should be long enough to prevent unwanted reactions to temporary switches (a cloud, a traffic light, ...).

Regarding modality changes, it is important to prevent continuous changes in the interaction mode as users can be irritated. In this case, user feedback mechanisms about the allowed interaction mode or feedback about upcoming modality changes can be helpful to improve the user experience.

Likewise, as already presented in [ABY13], task-feature set changes have a big potential to simplify the user interface. Regarding this adaptation category, it is essential to identify the right amount of information that is displayed on the screen, based on the current context-of-use.

Moreover, the usability evaluation results show that layout changes, in general, are useful to improve user satisfaction. However, as presented in [RB11] it is important to identify the individual and cultural preferences of the users as some adaptation rules were liked by a subset of the users and disliked by another subset of users.

Beside layout changes, the design of adaptive UIs can be improved when UI adaptations complement domain-specific functionalities of the targeted application.

Finally, based on the usability evaluation results, we identified that a smart composition of previously mentioned basic UI adaptations to a combined UI adaptation has the largest potential to improve user satisfaction. An important and still open question regarding this aspect is how to combine different basic adaptation rules to a smart combined adaptation rule that satisfies the current context-of-use the most.

Limitations of Usability Study

In this section, we shortly discuss the limitations of the conducted usability study and their impact on the validity of the results. Above all else, the small number of participants (and the low share of negative user responses) make it infeasible to draw any statistically significant conclusions from the experiment. In view of this fact, we are very careful to not make any definite claims regarding the interpretation of the results. Another limitation concerns the recruiting process of the participants of the experiment. The invitations were mostly sent to students, colleagues, and friends. It must be assumed that the invited students and colleagues are (compared to the general population) rather tech-savvy. In addition, all groups are likely well disposed towards the organizers, potentially leading to a bias of the collected feedback data. With regard to a potential bias of the collected data, we have to underline the fact that users can ignore feedback questions and that the absence of explicit user feedback should not be interpreted as a positive result for end-user satisfaction. Therefore, the resulting percentage values should be rather seen as an indicator. Finally, an open point for discussion is the frequency of asking users for explicit feedback. Asking for feedback in this fine-grained manner can annoy the users and result in an intrusive evaluation method. However, it can be seen as a trade-off where one has to decide about the frequency of asking users and the amount of qualitative user feedback data that can be collected in addition to implicit interaction data. Although our informal interview with the participants showed that the separate design of the screen (feedback bar and "task specific" UI) mitigate the mentioned problem of annoyance to some extent, there is still room for improvement regarding this aspect.

7.3 Summary and Discussion

In this chapter, we described two case studies for which we applied our model-driven development approach to devise self-adaptive UIs and presented a usability study to evaluate their acceptance by the end-users.

Regarding the case studies, we first introduced the evaluation questions for analyzing the applicability of our model-driven development approach. Then, we presented the two case studies showing the development of self-adaptive UIs for different application scenarios. The first application scenario for which we devised self-adaptive UIs was a library web application. In the second application scenario, analogously, we developed an e-mail application with context management and UI adaptation capabilities. For both case studies, the actual application of our model-driven development approach was described by introducing the main artifacts and findings of each development phase. The case study applicability evaluation was concluded with a summary and discussion of the evaluation questions.

Finally, an on-the-fly usability evaluation solution was presented. It was applied for the mentioned e-mail application to conduct a usability study which aimed to analyze the end-user satisfaction of self-adaptive UIs. After that, the conducted usability experiment and its main results were presented. Finally, we have discussed lessons learned and limitations of the conducted usability study.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented our solution for the model-driven engineering of self-adaptive user interfaces. Our solution approach covers modeling, transformation, execution, and evaluation of self-adaptive user interfaces. We presented a prototypical implementation of our model-driven engineering approach as well as its evaluation based on example application scenarios and a usability study. In this concluding chapter, we first summarize our contributions in Section 8.1. In Section 8.2, we then discuss how our solution fulfills the requirements identified in Section 3.2. An overview of future work in the field of model-driven engineering of self-adaptive UIs is given in Section 8.3.

8.1 Summary of Contributions

The development and evaluation of self-adaptive UIs is a challenging and complex task. Concerning development, aspects such as context management and UI adaptation further increase complexity compared to development of classical UIs and require an integrated development approach. Concerning evaluation, usability plays a crucial role for acceptance of self-adaptive UIs.

The goal of this thesis was to develop a solution for engineering self-adaptive user interfaces that support automatic UI adaptations at run-time as a reaction to dynamic context changes regarding user, platform, and environment characteristics. On the one hand, such a solution needs to support the development process by easing the developer's work in mastering the complex task of developing self-adaptive UIs. On the other hand, the outcome of the development process should result in self-adaptive UIs that are easy to use and offer a high usability for the end-users of the involved interactive system. To achieve the goals of this thesis, we presented our model-driven engineering approach for self-adaptive UIs that comprises four phases: *Modeling*, *Transformation*, *Execution*, and *Evaluation*. In the following, we shortly describe these contributions:

Modeling
The first phase of our approach supports the modeling of self-adaptive UIs. Based on the OMG's standardized UI modeling language *IFML*, we developed and introduced complementary domain-specific languages to cover the aspects of context-management and UI adaptation, which are important prerequisites for self-adaptive UIs. In this course, we have developed on the one hand *ContextML*, a textual modeling language, that supports the specification of context models that represent various context-of-use situations covering different static and dynamic aspects regarding user, platform, and environment. On the other hand, we have developed *AdaptML*, also a textual modeling language, that supports the specification of UI adaptation rules which describe how the UI is changed under which situation or circumstance. The introduced modeling languages *ContextML* and *AdaptML* have been integrated in the *IFML* Eclipse plugin to provide an integrated modeling workbench for self-adaptive UIs.

Transformation
Following the idea of model-driven development techniques, our solution provides three types of specific code generators to transform the specified models based on *IFML*, *ContextML*, and *AdaptML* into executable code artifacts. For this purpose, we have developed a new *UI Generator* that supports the transformation of final UI code from *IFML* models. In addition to that, we have developed a novel approach to derive executable *Context Services* from a context model. Our introduced *Context Service Generator* gets as input the specified context model and generates a Context Service that monitors the specified context properties through sensor probes. Analogously, we have developed an *Adaptation Service Generator* that gets as input the specified adaptation model and generates an *Adaptation Service* which is responsible for UI adaptation at runtime.

Execution
The execution of the self-adaptive UI is supported through an integrated UI framework where the different generated code artifacts regarding *Final UI* code, *Context Service*, and *Adaptation Service* are integrated in an overall execution environment. For supporting the monitoring and adaptation concerns at runtime, we have developed a run-time architecture for self-adaptive UIs. For implementing the rule-based execution environment we used the *Nools* rule engine.

Evaluation
The evaluation of our model-driven engineering approach consists of two parts. Firstly, the benefit of our model-driven engineering approach is demonstrated by two case-studies showing the development of self-adaptive UIs for different application scenarios. The first application scenario targets a library web application while the second one deals with an e-mail client application for which we devised self-adaptive UIs. Furthermore, we have also evaluated the usability of self-adaptive UIs by conducting a usability study. For this purpose, we have analyzed the usability of an adaptive mobile e-mail application. The adaptive mobile e-mail application contains different adaptation features that are triggered at runtime and can be rated by the users. Based on the stored context information and gathered user feedback, we have conducted a data-driven usability evaluation of self-adaptive UIs regarding end-user satisfaction.

8.2 Requirements Revisited

In Section 3.2, we stated a set of requirements that a model-driven engineering approach needs to fulfill in order to enable the development and evaluation of self-adaptive user interfaces. Subsequently, we describe how our solution fulfills these requirements.

Context Management:
The requirements for context management claim that the solution approach for engineering self-adaptive UIs supports context modeling (R1), context model transformation (R2), runtime context monitoring (R3), and appropriate tool support for managing those context management activities (R4).

Our solution approach introduces a context modeling language, *ContextML*, which supports the specification of various context-of-use parameters. Thus, requirement (R1) is fulfilled. The second requirement, context transformation (R2), is addressed by our *Context Service Generator* which has been implemented to support the transformation of context models based on *ContextML* to executable code of *Context Services*. As the generated *Context Services* enable continuous monitoring of heterogeneous context-of-use parameters at runtime though different hardware sensors, requirement (R3) is also fulfilled. Finally, appropriate tool support for context management (R4) is also given by our solution approach as it provides a dedicated, modeling view for context modeling and a *Context Service Generator* in the proof-of-concept implementation.

UI Adaptation:
The requirements for UI adaptation claim that the solution approach for engineering self-adaptive UIs supports adaptation modeling (R5), adaptation model transformation (R6), runtime UI adaptation (R7) and appropriate tool support for managing those UI adaptation activities (R8).

Our solution approach introduces the adaptation modeling language *AdaptML*, which supports the specification of various UI adaptation rules covering different adaptation techniques. Thus, requirement (R5) is fulfilled. The next requirement, adaptation model transformation (R6) is addressed by our *Adaptation Service Generator* which has been implemented to support the transformation of adaptation models based on *AdaptML* to executable code of adaptation rules that are managed by the rule engine *Nools*. As the generated *Adaptation Services* enable continuous runtime UI adaptation, requirement (R7) is also fulfilled. Finally, appropriate tool support for UI adaptation (R8) is also given by our solution approach as it provides a dedicated, adaptation modeling view and an *Adaptation Service Generator* in the proof-of-concept implementation.

Usability Evaluation:
The requirements for usability evaluation of self-adaptive UIs claim that the solution approach for assessing the quality of use of such UIs considers end-user satisfaction (R9), incorporates a collection of explicit and implicit user-feedback (R10), and enables a runtime usability testing (R11) of the UI while end-users are interacting with the systems and UI adaptations are happening at runtime.

Our on-the-fly usability evaluation solution focuses on end-user satisfaction analysis as we ask the end-users to rate the triggered UI adaptations (like or dislike). Thus, we get explicit feedback from the end-users whether they appreciate the triggered UI adaptations or not, and thus (R9) is fulfilled in our solution. As the end-users are asked for feedback whenever a UI adaptation feature is triggered at runtime, also (R10) is fulfilled, because based on the very current context-of-use we get and map to it the user feedback. Finally, requirement (R11) is also addressed by our approach, as the usability evaluation method enables evaluation of the acceptance of UI adaptations while the end-users are interacting with the interactive system at the moment when UI adaptations occur and the end-user's indentation about them are quite fresh.

As described in this section, our solution for model-driven engineering of self-adaptive UIs fulfills the requirements stated in Section 3.2. As a conclusion of this, the research question, introduced in the beginning of this thesis has been addressed within our model-driven engineering approach for self-adaptive UIs.

8.3 Future Work

In this thesis, we presented a model-driven engineering approach for self-adaptive UIs. Apart from its current features and contributions, there are certain aspects that can be addressed in future work to further advance the solution approach. In this section, based on the phases of our engineering approach, we discuss possible extensions and present ideas for follow-up research.

Modeling Extensions
Our model-driven engineering approach aims to support the development of self-adaptive UIs in a comprehensive way. For this purpose, we address the key modeling concerns UI, context management, and UI adaptation in our solution approach. However, the modeling and generation of application logic was out of scope and is only partially addressed through *IFML Action* elements. Thus, we only generate function stubs for the application logic that still must be manually coded. To overcome this issue, additional models such as BPMN or UML state charts, activity or sequence diagrams could be used to cover application logic aspects and to integrate them into the overall engineering approach for self-adaptive UIs.

While our introduced textual modeling languages *ContextML* and *AdaptML* provide advantages such as auto-completion and syntax checks in specifying the context and adaptation model, it would be helpful for UI/Web designers to have a visual language that supports the modeling of these aspects in a similar graphical way like the modeling of the core UI based on *IFML*. Moreover, it has to be noticed that the introduced modeling languages *ContextML* and *AdaptML* provide a core basis to support holistic modeling of various context-of-use parameters and UI adaptation rules. Although the metamodels provide predefined classes for characterizing various aspects, they do not have the objective to be complete, but they are rather extensible for covering further aspects. As an example, the metamodel *UserContext* provides only basic predefined classes for characterizing the end-users of a self-adaptive UI. Of course, one can think about more fine grained modeling techniques to characterize the user, for example, through a digital twin of a human model as we have presented in [JYE19b] for supporting multi-modal UI adaptation for assistance systems in Industry 4.0 scenarios [JYE19a]. Moreover, considering the presented case studies in the previous chapter, we observed that our introduced domain specific languages *ContextML* and *AdaptML* are a suitable complement to the OMG's UI modeling language *IFML* to specify context management and

UI adaptation concerns. Especially the separation of different modeling views for UI, context, and adaptation eases the modeling of self-adaptive UIs and also supports the maintenance of evolving context and adaptation models. As the presented case studies are showing, our integrated development approach allows the generation of self-adaptive UIs that can have quite complex UI adaptation features. In this regard, we have to point out that our approach is not supporting the analysis and resolution of conflicting UI adaptation rules. Although we have introduced priority levels to determine the execution order of UI adaptation rules, still it is a complex and error-prone task to manually specify a sound set of UI adaptations. When designing the UI adaptation rules it can happen due to human errors that conflicting adaptation rules are modeled which can result in undesirable UI adaptations. To solve this problem further formalization of adaptation rules for example based on TGGs is possible to establish analysis techniques to detect conflicting rules already at design-time [AYKP18]. As an extension of this idea, in [AYK19], we suggest a useful definition of consistency for runtime UI adaptation, identify some important and open consistency-related challenges, and highlight solution strategies inspired by results and in-sights from research on bidirectional transformations.

Transformation Extensions

Regarding the transformation approach, our provided code generators have not the objective of completeness, but rather show for a subset of *IFML* models that the generation of code for self-adaptive UIs can be automated.

The proof-of-concept implementation targets the generation of UI views based on the Angular framework. Although Angular is a widespread and commonly used UI framework for web applications targeting various end-devices, other UI frameworks such as Vaadin[1], Bootstrap[2] or React[3] could be used as well for covering the UI part of the self-adaptive UI.

Regarding the generation of *Context Services*, due to the very individual structure of some context sources, the code for controlling and managing sensor sources, like APIs, SDKs or libraries, could not be always automatically generated and thus had to be implemented manually. The code is written in static files, which are scanned during the generation process and inserted into predefined sections among the generated code. Hence, the proof-of-concept implementation of the *Context Service Generator* covers the automated generation of code for a specific set of *Context Services* and can be extended to cover further context sensors and information.

[1]https://vaadin.com
[2]https://getbootstrap.com
[3]https://reactjs.org

Our *Adaptation Service* generation approach relies on the usage of *Nools* which is an efficient rule engine based on the RETE algorithm. In cases where other rule engines are required, the flexible structure of our *Adaptation Service Generator* can be easily adjusted according to new code templates to enable a translation of the *AdaptML* models to the syntax of the target rule engine.

Execution Extensions

Regarding the execution phase it is still a challenging task to correctly time the adaptation operations as time delays in the communication might happen. For example, the camera is used for sensing the mood of the user and it takes time to get the this information from the face detection API from a different server. Also, in some cases it is needed to undo triggered UI adaptation operations which is currently not supported in the actual version of the execution environment due to dependencies between adaptation operations. However, in our implementation of the execution environment, the end-user has control over the adaptation process and can decide whether the monitoring and adaptation processes are switched on or off. As mentioned above, in our presented engineering approach we use priority levels to define the execution sequence of UI adaptation rules and to prevent conflicting adaptation rules. However, beside the described strategies to tackle conflicting rules at design time, there are further improvement possibilities regarding the quality assurance of self-adaptive UIs at runtime. In this context, it is possible to use model-checking techniques to verify the correctness of UI adaptation operations in order to guarantee termination and to prevent oscillation of UI adaptations. As the complexity of the UI adaptation logic is increasing with new adaptation features, it is also possible to think about software testing approaches (especially model-based software testing) to ensure correctness, proper functionality of the Adaptation Features under dynamically changing context-of-use situations. Finally, it would be very beneficial to have a large-scale evaluation of the generated self-adaptive UIs to further analyze the runtime scalability aspect. With this regard, further application of our engineering approach in different domains would be helpful to gather more information for improving the execution environment.

Evaluation Extensions

By the evaluation performed in this thesis, we were able to demonstrate the applicability of our engineering approach to develop self-adaptive UIs for different application scenarios. Further evaluation of the introduced modeling languages *ContextML* and *AdaptML* regarding usability or learnability can be conducted to gain further insights about the acceptance and productivity of the developers. In this context, the whole model-driven engineering approach for self-adaptive UIs can be evaluated in detail regarding effectiveness and efficiency to better analyze its strength and weaknesses from a developer's perspective.

In our solution, we rather focused on the usability evaluation of the self-adaptive UIs resulting from our model-driven engineering approach. For this purpose, we have conducted a data-driven usability evaluation experiment with real end-users to analyze the quality and acceptance of UI adaptations in different context-of-use situations. Regarding the usability evaluation results, we can sum up that most of the triggered UI adaptation features were positively rated. Although this is a positive indicator for the resulting self-adaptive UIs, it should be noticed that users can ignore feedback questions in our usability experiment setup and that the absence of explicit user feedback should not be interpreted as a positive result for end-user satisfaction in general. Moreover, an important observation from the usability study was that some of the users feel disturbed by the UI adaptations when they are not aware about the changes. An improvement in this directions would be to integrate smooth animations which show the UI adaptation changes during the adaptation process [DMV11].

Future experiments, especially long-term evaluation studies should be done with a larger group of participants to collect even more data about the context-of-use and instant user feedback to support even stronger evidence for our findings. We are confident that given the data of a larger user test group, it is possible to identify statistical significant interrelations between the context-of-use and the user feedback when using more sophisticated techniques such as regression analysis. Future experiments should also cover a direct comparison between the adaptive and non-adaptive version of the UI.

Beside end-user satisfaction, further usability criteria such as effectiveness or efficiency could be also assessed to gain broader insights about the resulting self-adaptive UIs.

For future work, we have several ideas on how to extend the approach to improve the solution for evaluating self-adaptive UIs and, as a consequence, how to improve the quality of UI adaptation features. Firstly, a mutation analysis approach can be integrated in our solution to intersperse "bad UI adaptations", so called mutants in the application. This way, a more objective rating from end-user perspective can be reached, as they are not always shown useful adaptation features. In addition, it is possible to use implicit feedback mechanisms (e.g., shutdown of app, mood, usage time etc.) which are already tracked in our approach to further improve UI adaptations. In some cases, dependent on the application domain, such as in the presented mail application, it is also possible to adapt the UI based on the content of the UI elements, like the mail content itself.

While collecting context information, also issues regarding data privacy can arise and should be considered. On the one hand, a fine-grained way of collecting explicit instant user feedback can annoy the users and result in an intrusive evaluation method [YHR+19]. On the other hand, the collection of instant user feedback can be used to further optimize UI adaptations (see next point).

Follow-Up Work: Smart self-adaptive UIs based on machine learning:
In its current state, the implemented adaptation process follows a rule-based approach and relies on a rule engine. Further optimization of UI adaptations can be reached through extending the adaptation manager by machine learning algorithms. This way, log data (context information, previous adaptations, and user feedback) can be analyzed to learn the most suitable adaptations for future context-of-use situations. Based on our data-driven usability experiment with an adaptive mobile app, we have established the main prerequisites to monitor and store context information as well as collect feedback from end-users. This basis can be used to extend our approach with further machine learning techniques to realize more 'intelligent' self-adaptive UIs.

Follow-Up Work: From Legacy UIs to Self-adaptive UIs:
As presented in this thesis, our model-driven engineering approach for self-adaptive UIs starts from scratch where developers need to create the relevant UI, context, and adaptation models. Based on the specified models, our engineering approach supports the automated generation of self-adaptive UIs. While this approach is especially helpful for new application scenarios starting on the green field or software systems including relevant UI, context, and adaptation models, there are also lots of software systems that do neither rely on such models nor incorporate UI adaptation capabilities. In most cases it could be beneficial to integrate self-adaptive UIs in such legacy software systems especially when it comes to UI modernization. Indeed this idea has been already presented in [ABY14b], where the incorporation of adaptive UIs is shown based on basic UI adaptation features. However, this idea can be extend in such a way that more UI adaptations are incorporated to enable sophisticated UI adaptations. To tackle this issue and to support an automatic transition of non-adaptive legacy UIs to self-adaptive UIs, in [JYS18], we have addressed the reverse engineering process to enable the extraction of *IFML* models from legacy UIs. Once the *IFML* model of the legacy system's UI is extracted, we restructure and enrich it with self-adaptivity features by applying our existing forward engineering approach for self-adaptive UIs. In addition to that, techniques from process mining could be used to identify behavioral usage patterns (interaction logs) for different individual users to better guide the context management and UI adaptation process.

Follow-Up Work: Context-aware AR/VR:
The idea of self-adaptive UIs comprising context management and UI adaptation can be applied beyond classical desktop, web, and mobile applications. With the spread of augmented (AR) and virtual reality (VR) applications in various domains (e.g., education and training, engineering or entertainment) targeting heterogeneous end-users, target platforms (VR headsets, AR glasses, Mobile AR etc.), and dynamically changing mixed environments (merging virtual and real objects as introduced in mixed reality interfaces) the aspect *context-awareness* started to play an important role. Therefore, we have started to apply and transfer existing concepts of this thesis to virtual and augmented reality interfaces. In our recent work [YHE19], we have presented a framework for supporting context-aware virtual reality applications. The benefit of this framework was shown based on a first aid training application and VR demo. Furthermore, the same idea was also applied for augmented reality applications. Firstly in [YJSE19b], we have identified the main challenges for development of context-aware AR applications. To address these challenges, in [YJSE19a] we have introduced the idea of a model-based framework for the development of context-aware AR applications. To show the advantage of such context-aware applications we have instantiated example AR applications for maintenance applications. In future work, we will further investigate the potentials of context-awareness and UI adaptation for mixed reality interfaces in the context of different application domains such as cross-device mixed reality interfaces or human-robot/human-human collaboration via AR/VR.

References

[ABY12] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Using interpreted runtime models for devising adaptive user interfaces of enterprise applications. In *ICEIS 2012 - Proceedings of the 14th International Conference on Enterprise Information Systems, Volume 3, Wroclaw, Poland, 28 June - 1 July, 2012*, pages 72–77, 2012.

[ABY13] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. RBUIS: simplifying enterprise application user interfaces through engineering role-based adaptive behavior. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'13, London, United Kingdom - June 24 - 27, 2013*, pages 3–12, 2013.

[ABY14a] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Adaptive model-driven user interface development systems. *ACM Comput. Surv.*, 47(1):9:1–9:33, 2014.

[ABY14b] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Integrating adaptive user interface capabilities in enterprise applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 712–723, 2014.

[ABY16] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Engineering adaptive model-driven user interfaces. *IEEE Trans. Software Eng.*, 42(12):1118–1147, 2016.

[AIV08] Silvia Abrahão, Emilio Iborra, and Jean Vanderdonckt. Usability evaluation of user interfaces generated with a model-driven architecture tool. In *Maturing Usability - Quality in Software, Interaction and Value*, pages 3–32. Springer, 2008.

[Aki14] Pierre A. Akiki. *Engineering adaptive model-driven user interfaces for enterprise applications*. PhD thesis, Open University, UK, 2014.

[AMI17] Mai Abusair, Antinisca Di Marco, and Paola Inverardi. Context-aware adaptation of mobile applications driven by software quality and user satisfaction. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017, Prague, Czech Republic, July 25-29, 2017*, pages 31–38, 2017.

[APB+99] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: an appliance-independent XML user interface language. *Computer Networks*, 31(11-16):1695–1708, 1999.

[AYG10] Achilleas Achilleos, Kun Yang, and Nektarios Georgalas. Context modelling and a context-aware framework for pervasive service creation: A model-driven approach. *Pervasive and Mobile Computing*, 6(2):281–296, 2010.

[AYK19] Anthony Anjorin, Enes Yigitbas, and Hermann Kaindl. Consistent runtime adaptation of user interfaces. In *Proceedings of the 8th International Workshop on Bidirectional Transformations co-located with the Philadelphia Logic Week, BxPLW 2019, Philadelphia, PA, USA, June 4, 2019.*, pages 61–65, 2019.

[AYKP18] Anthony Anjorin, Enes Yigitbas, Hermann Kaindl, and Roman Popp. On the development of consistent user interfaces (extended abstract). In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, pages 18–20, 2018.

[Bar05] Jakob E. Bardram. The java context awareness framework (JCAF) - A service infrastructure and programming framework for context-aware applications. In *Pervasive Computing, Third International Conference, PERVASIVE 2005, Munich, Germany, May 8-13, 2005, Proceedings*, pages 98–115, 2005.

[BB10] Arnaud Blouin and Olivier Beaudoux. Improving modularity and usability of interactive systems with malai. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2010, Berlin, Germany, June 19-23, 2010*, pages 115–124, 2010.

[BCM07] Andrea Bunt, Cristina Conati, and Joanna McGrenere. Supporting interface customization using a mixed-initiative approach. In *Proceedings of the 12th International Conference on Intelligent User Interfaces, IUI 2007, Honolulu, Hawaii, USA, January 28-31, 2007*, pages 92–101, 2007.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool Publishers, 1st edition, 2012.

[BDB+04] Lionel Balme, Alexandre Demeure, Nicolas Barralon, Joëlle Coutaz, and Gaëlle Calvary. CAMELEON-RT: A software architecture reference model for distributed, migratable, and plastic user interfaces. In *Ambient Intelligence: Second European Symposium, EUSAI 2004, Eindhoven, The Netherlands, November 8-11, 2004. Proceedings*, pages 291–302, 2004.

[BF14] Marco Brambilla and Piero Fraternali. *Interaction Flow Modeling Language - Model-Driven UI Engineering of Web and Mobile Apps with IFML.* The MK/OMG Press, 2014.

[BHV+94] Francois Bodart, Anne-Marie Hennebert, Leheureux Jean Vanderdonckt, Jean Vanderdonckt, Franois Bodart, Anne marie Hennebert, Jean marie

Leheureux, Isabelle Provot, and Jean V. A model-based approach to presentation: A continuum from task analysis to prototype. In *In Proceedings of DSV-IS'94*, pages 25–39. Springer-Verlag, 1994.

[BJM+05] Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen. *Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BMB+11] Arnaud Blouin, Brice Morin, Olivier Beaudoux, Grégory Nain, Patrick Albers, and Jean-Marc Jézéquel. Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011*, pages 85–94, 2011.

[Bot11] Goetz Botterweck. Multi front-end engineering. In *Model-Driven Development of Advanced User Interfaces*, pages 27–42. 2011.

[BUA17] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. Model-driven development of user interfaces for iot systems via domain-specific components and patterns. *J. Internet Services and Applications*, 8(1):14:1–14:21, 2017.

[CCD+04] Gaëlle Calvary, Joëlle Coutaz, Olfa Dâassi, Lionel Balme, and Alexandre Demeure. Towards a new generation of widgets for supporting software plasticity: The "comet". In *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers*, pages 306–324, 2004.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS*, 15:289–308, 2003.

[CDMF07] Stefano Ceri, Florian Daniel, Maristella Matera, and Federico Michele Facca. Model-driven development of context-aware web applications. *ACM Trans. Internet Techn.*, 7(1):2, 2007.

[CFB00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[CLV+03] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In *Human-Computer Interaction with Mobile Devices and Services, 5th International Symposium, Mobile HCI 2003, Udine, Italy, September 8-11, 2003, Proceedings*, pages 256–270, 2003.

[Cou10] Joëlle Coutaz. User interface plasticity: Model driven engineering to the limit! In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 1–8, New York, NY, USA, 2010. ACM.

[DAS01] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.

[DC06] Carlos Duarte and Luís Carriço. A conceptual framework for developing adaptive multimodal applications. In *Proceedings of the 11th International Conference on Intelligent User Interfaces, IUI 2006, Sydney, Australia, January 29 - February 1, 2006*, pages 132–139, 2006.

[DCC08] Alexandre Demeure, Gaëlle Calvary, and Karin Coninx. Comet(s), A software architecture style and an interactors toolkit for plastic user interfaces. In *Interactive Systems. Design, Specification, and Verification, 15th International Workshop, DSV-IS 2008, Kingston, Canada, July 16-18, 2008, Revised Papers*, pages 225–237, 2008.

[Dey01] Anind K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, January 2001.

[dK01] Nora Parcus de Koch. *Software engineering for adaptive hypermedia systems: reference model, modeling techniques and development process*. PhD thesis, Ludwig Maximilians University Munich, 2001.

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat: a generic framework for context-aware applications. In *Proceedings of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC 2005), held at the ACM/IFIP/USENIX 6th International Middleware Conference, November 28 - December 2, 2005, Grenoble, France*, pages 1–7, 2005.

[DMV11] Charles-Eric Dessart, Vivian Genaro Motti, and Jean Vanderdonckt. Showing user interface adaptivity by animated transitions. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2011, Pisa, Italy, June 13-16, 2011*, pages 95–104, 2011.

[FBA06] Sebastian Feuerstack, Marco Blumendorf, and Sahin Albayrak. Bridging the gap between model and design of user interfaces. In *Informatik 2006 - Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*, pages 131–137, 2006.

[FBK$^+$08] Sebastian Feuerstack, Marco Blumendorf, Maximilian Kern, Michael Kruppa, Michael Quade, Mathias Runge, and Sahin Albayrak. Automated usability evaluation during model-based interactive system development. In *Engineering Interactive Systems, Second Conference on Human-Centered Software Engineering, HCSE 2008, and 7th International Workshop on Task Models and Diagrams, TAMODIA 2008, Pisa, Italy, September 25-26, 2008. Proceedings*, pages 134–141, 2008.

[FG09] Leah Findlater and Krzysztof Z. Gajos. Design space and evaluation challenges of adaptive graphical user interfaces. *AI Magazine*, 30(4):68–73, 2009.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[FRYF16] Holger Fischer, Mirko Rose, Enes Yigitbas, and Peter Forbrig. Towards a task driven approach enabling continuous user requirements engineering. In *Joint Proceedings of REFSQ-2016 Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 22nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016), Gothenburg, Sweden, March 14, 2016.*, 2016.

[FYS15] Holger Fischer, Enes Yigitbas, and Stefan Sauer. Integrating human-centered and model-driven methods in agile UI development. In *Proceedings of 15th IFIP TC.13 International Conference on Human-Computer Interaction (INTERACT)*, volume 22, pages 215–221. University of Bamberg Press, 2015.

[GET$^+$08] Krzysztof Z. Gajos, Katherine Everitt, Desney S. Tan, Mary Czerwinski, and Daniel S. Weld. Predictability and accuracy in adaptive user interfaces. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI 2008, 2008, Florence, Italy, April 5-10, 2008*, pages 1271–1274, 2008.

[GMP$^+$15] Giuseppe Ghiani, Marco Manca, Fabio Paternò, Jörg Rett, and Atul Vaibhav. Adaptive multimodal web user interfaces for smart work environments. *JAISE*, 7(6):701–717, 2015.

[GS01] Philip D. Gray and Daniel Salber. Modelling and using sensed context information in the design of interactive applications. In *Engineering for Human-Computer Interaction, 8th IFIP International Conference, EHCI 2001, Toronto, Canada, May 11-13, 2001, Revised Papers*, pages 317–336, 2001.

[GS13] Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly Media, Inc., 1st edition, 2013.

[GWW10] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. Automatically generating personalized user interfaces with supple. *Artif. Intell.*, 174(12-13):910–950, August 2010.

[Har16] Patrick Harms. *Automated Field Usability Evaluation Using Generated Task Trees*. PhD thesis, University of Göttingen, 2016.

[HBSS02] Albert Held, Sven Buchholz, Alexander Schill, and Er Schill. Modeling of context information for pervasive computing applications, 2002.

[HG14] Patrick Harms and Jens Grabowski. Usage-based automatic detection of usability smells. In *Human-Centered Software Engineering - 5th IFIP WG 13.2 International Conference, HCSE 2014, Paderborn, Germany, September 16-18, 2014.*, pages 217–234, 2014.

[HHB$^+$18] Jamil Hussain, Anees Ul Hassan, Hafiz Syed Muhammad Bilal, Rahman Ali, Muhammad Afzal, Shujaat Hussain, Jae Hun Bang, Oresti Banos, and Sungyoung Lee. Model-based adaptive user interface based on context and user experience evaluation. *J. Multimodal User Interfaces*, 12(1):1–16, 2018.

[HIR03] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Generating context management infrastructure from high-level context models. In *In 4th International Conference on Mobile Data Management (MDM) - Industrial Track*, pages 1–6, 2003.

[HS09] Christian Hoareau and Ichiro Satoh. Modeling and processing information for context-aware computing: A survey. *New Generation Comput.*, 27(3):177–196, 2009.

[HTLK08] Anas Hariri, Dimitri Tabary, Sophie Lepreux, and Christophe Kolski. Context aware business adaptation toward user interface adaptation. In *In Communications of SIWN*, pages 46–52. Springer Verlag, 2008.

[IBM05] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[JDB18] Imen Jaouadi, Raoudha Ben Djemaa, and Hanêne Ben-Abdallah. A model-driven development approach for context-aware systems. *Software and System Modeling*, 17(4):1169–1195, 2018.

[JYE19a] Klementina Josifovska, Enes Yigitbas, and Gregor Engels. A digital twin-based multi-modal UI adaptation framework for assistance systems in industry 4.0. In *Human-Computer Interaction. Design Practice in Contemporary Societies - Thematic Area, HCI 2019, Held as Part of the 21st HCI International Conference, HCII 2019, Orlando, FL, USA, July 26-31, 2019, Proceedings, Part III*, pages 398–409, 2019.

[JYE19b] Klementina Josifovska, Enes Yigitbas, and Gregor Engels. Reference framework for digital twins within cyber-physical systems. In *Proceedings of the 5th International Workshop on Software Engineering for Smart Cyber-Physical Systems, SEsCPS at ICSE 2019, Montreal, QC, Canada, May 28, 2019.*, pages 25–31, 2019.

[JYS18] Ivan Jovanovikj, Enes Yigitbas, and Stefan Sauer. Model-based ui modernization: From legacy uis to self-adaptive uis. *Softwaretechnik-Trends, Proceedings of the 20th Workshop Software-Reengineering and Evolution (WSRE) and 9th Workshop Design for Future (DFF)*, 2018.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KSR03] Laurie Kantner, Deborah Hinderer Sova, and Stephanie Rosenbaum. Alternative methods for field usability research. In *Proceedings of the 21st annual international conference on Documentation, SIGDOC 2003, San Francisco, CA, USA, October 12-15, 2003*, pages 68–72, 2003.

[Lad97] R. Laddaga. Self-adaptive software. Technical Report 98-12, DARPA BAA, 1997.

[LM10] Talia Lavie and Joachim Meyer. Benefits and costs of adaptive user interfaces. *Int. J. Hum.-Comput. Stud.*, 68(8):508–524, 2010.

[LP08] Gitte Lindgaard and Avi Parush. Utility and experience in the evolution of usability. In *Maturing Usability - Quality in Software, Interaction and Value*, pages 222–240. Springer, 2008.

[LRBA10] G. Lehmann, A. Rieger, M. Blumendorf, and S. Albayrak. A 3-layer architecture for smart environment models. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 636–641, March 2010.

[LVM$^+$04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. USIXML: A language supporting multi-path development of user interfaces. In *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers*, pages 200–220, 2004.

[Mez13] Nesrine Mezhoudi. User interface adaptation based on user feedback and machine learning. In *18th International Conference on Intelligent User Interfaces, IUI '13, Santa Monica, CA, USA, March 19-22, 2013, Companion Volume*, pages 25–28, 2013.

[MM02] Nikola Mitrovic and Eduardo Mena. Adaptive user interface for mobile devices. In *Interactive Systems. Design, Specification, and Verification, 9th International Workshop, DSV-IS 2002, Rostock Germany, June 12-14, 2002*, pages 29–43, 2002.

[MM03] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical Report omg/03-06-01, Object Management Group (OMG), June 2003.

[Mot13] Vivian Genaro Motti. *TriPlet : a conceptual framework for multidimensional adaptation of user interfaces to the context of use*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2013.

[MPV11] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10:2–11, 2011.

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM.

[MV13] Vivian Genaro Motti and Jean Vanderdonckt. A computational framework for context-aware adaptation of user interfaces. In *IEEE 7th International Conference on Research Challenges in Information Science, RCIS 2013, Paris, France, May 29-31, 2013*, pages 1–12, 2013.

[Neb12] Michael Nebeling. *Lightweight informed adaptation: Methods and tools for responsive design and development of very flexible, highly adaptive web interfaces*. PhD thesis, ETH ZURICH, 2012.

[Nie93] Jakob Nielsen. *Usability engineering*. Academic Press, 1993.

[NSN13] Michael Nebeling, Maximilian Speicher, and Moira C. Norrie. Crowdadapt: enabling crowdsourced web page adaptation for individual viewing conditions and preferences. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'13, London, United Kingdom - June 24 - 27, 2013*, pages 23–32, 2013.

[Obj15] Object Management Group (OMG). Interaction Flow Modeling Language (IFML) Specification, Version 1.0. OMG Document Number formal/2015-02-05 (https://www.omg.org/spec/IFML/1.0/PDF), 2015.

[Obj17] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 2.5.1. OMG Document Number formal/2017-12-05 (https://www.omg.org/spec/UML/2.5.1/PDF), 2017.

[OGT$^+$99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[Pat13] Fabio Paternò. User interface design adaptation. In *The Encyclopedia of Human-Computer Interaction, 2nd Ed.*, chapter 39. Soegaard, Mads and Dam, Rikke Friis (eds.)., Aarhus, Denmark, 2013.

[Pet07] Roland Petrasch. Model based user interface design: Model driven architecture und HCI patterns TM. *Softwaretechnik-Trends*, 27(3), 2007.

[PHJS12] Matthias Peissner, Dagmar Häbe, Doris Janssen, and Thomas Sellner. Myui: generating accessible user interfaces from multimodal design patterns. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, Copenhagen, Denmark - June 25 - 28, 2012*, pages 81–90, 2012.

[PLN04] Tim F. Paymans, Jasper Lindenberg, and Mark A. Neerincx. Usability trade-offs for adaptive user interfaces: ease of use and learnability. In *Proceedings of the 9th International Conference on Intelligent User Interfaces, IUI 2004, Funchal, Madeira, Portugal, January 13-16, 2004*, pages 301–303, 2004.

[PM15] Philippe A. Palanque and Célia Martinie. Designing and assessing interactive systems using task models. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems, Seoul, CHI 2015 Extended Abstracts, Republic of Korea, April 18 - 23, 2015*, pages 2465–2466, 2015.

[PMM97] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *Human-Computer Interaction, INTERACT '97, IFIP TC13 Interantional Conference on Human-Computer Interaction, 14th-18th July 1997, Sydney, Australia*, pages 362–369, 1997.

[PS12] Fabio Paternò and Carmen Santoro. A logical framework for multi-device user interfaces. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, Copenhagen, Denmark - June 25 - 28, 2012*, pages 45–50, 2012.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16(4):19:1–19:30, 2009.

[Pue97] A. R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4):40–47, Jul 1997.

[RB11] Katharina Reinecke and Abraham Bernstein. Improving performance, perceived usability, and aesthetics with culturally adaptive user interfaces. *ACM Trans. Comput.-Hum. Interact.*, 18(2):8:1–8:29, 2011.

[RMB13] David Raneburger, Gerrit Meixner, and Marco Brambilla. Platform-independence in model-driven development of graphical user interfaces for multiple devices. In *Software Technologies - 8th International Joint Conference, ICSOFT 2013, Reykjavik, Iceland, July 29-31, 2013, Revised Selected Papers*, pages 180–195, 2013.

[Sau11] Stefan Sauer. Applying meta-modeling for the definition of model-driven development methods of advanced user interfaces. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, pages 67–86. Springer, 2011.

[SAW94] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *First Workshop on Mobile Computing Systems and Applications, WMCSA 1994, Santa Cruz, CA, USA, December 8-9, 1994*, pages 85–90, 1994.

[SBG99] Albrecht Schmidt, Michael Beigl, and Hans-Werner Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893–901, 1999.

[SBM06] Robbie Schaefer, Steffen Bleul, and Wolfgang Müller. Dialog modeling for multiple devices and multiple interaction modalities. In *Task Models and Diagrams for Users Interface Design, 5th International Workshop, TAMODIA 2006, Hasselt, Belgium, October 23-24, 2006. Revised Papers*, pages 39–53, 2006.

[SCCF07] Jean-Sébastien Sottet, Gaëlle Calvary, Joëlle Coutaz, and Jean-Marie Favre. A model-driven engineering approach for the usability of plastic user interfaces. In *Engineering Interactive Systems - EIS 2007 Joint Working*

Conferences, EHCI 2007, DSV-IS 2007, HCSE 2007, Salamanca, Spain, March 22-24, 2007. Selected Papers, pages 140–157, 2007.

[Sch06] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.

[Sch13] Albrecht Schmidt. Context-aware computing. *The Encyclopedia of Human-Computer Interaction, 2nd Ed., Chapter 14*, 2013.

[Shn00] Ben Shneiderman. Universal usability. *Commun. ACM*, 43(5):84–91, 2000.

[SLP04] Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004.

[SOB14] Norbert Seyff, Gregor Ollmann, and Manfred Bortenschlager. Appecho: a user-driven, in situ feedback approach for mobile platforms and applications. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems, MOBILESoft 2014, Hyderabad, India, June 2-3, 2014*, pages 99–108, 2014.

[SRS15] Elhadi M. Shakshuki, Malcolm Reid, and Tarek R. Sheltami. An adaptive user interface in healthcare. In *The 10th International Conference on Future Networks and Communications (FNC 2015) / The 12th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2015) / Affiliated Workshops, August 17-20, 2015, Belfort, France*, pages 49–58, 2015.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[Sta98] International Organization For Standardization. *ISO 9241-11: Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs): Part 11: Guidance on Usability*. ISO, 1998.

[Sze96] Pedro Szekely. Retrospective and challenges for model-based interface development. In *Design, Specification and Verification of Interactive Systems '96*, pages 1–27. Springer-Verlag, 1996.

[TA13] Tom Tullis and Bill Albert. Chapter 9 - special topics. In Tom Tullis and Bill Albert, editors, *Measuring the User Experience (Second Edition)*, Interactive Technologies, pages 209 – 236. Morgan Kaufmann, Boston, second edition edition, 2013.

[Træ02] Hallvard Trætteberg. *Model-based User Interface Design*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2002.

[vVvdGKS08] Lex van Velsen, Thea van der Geest, Rob Klaassen, and Michaël F. Steehouder. User-centered evaluation of adaptive and adaptable systems: a literature review. *Knowledge Eng. Review*, 23(3):261–281, 2008.

[Wei99] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999.

[WSvT10] Janet Louise Wesson, Akash Singh, and Bradley van Tonder. Can adaptive interfaces improve the usability of mobile applications? In *Human-Computer Interaction - Second IFIP TC 13 Symposium, HCIS 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings*, pages 187–198, 2010.

[YAJ$^+$18] Enes Yigitbas, Anthony Anjorin, Ivan Jovanovikj, Thomas Kern, Stefan Sauer, and Gregor Engels. Usability evaluation of model-driven cross-device web user interfaces. In *Human-Centered Software Engineering - 7th IFIP WG 13.2 International Working Conference, HCSE 2018, Sophia Antipolis, France, September 3-5, 2018, Revised Selected Papers*, pages 231–247, 2018.

[YFKP14] Enes Yigitbas, Holger Fischer, Thomas Kern, and Volker Paelke. Model-based development of adaptive UIs for multi-channel self-service systems. In *Human-Centered Software Engineering - 5th IFIP WG 13.2 International Conference, HCSE 2014, Paderborn, Germany, September 16-18, 2014. Proceedings*, pages 267–274, 2014.

[YGS13] Enes Yigitbas, Christian Gerth, and Stefan Sauer. Konzeption modellbasierter benutzungsschnittstellen für verteilte selbstbedienungssysteme. In *Informatik 2013, 43. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Informatik angepasst an Mensch, Organisation und Umwelt, 16.-20. September 2013, Koblenz, Deutschland*, pages 2714–2723, 2013.

[YGSE17] Enes Yigitbas, Silas Grün, Stefan Sauer, and Gregor Engels. Model-driven context management for self-adaptive user interfaces. In *Ubiquitous Computing and Ambient Intelligence - 11th International Conference, UCAmI 2017, Philadelphia, PA, USA, November 7-10, 2017, Proceedings*, pages 624–635, 2017.

[YHE19] Enes Yigitbas, Joshua Heindörfer, and Gregor Engels. A context-aware virtual reality first aid training application. In *Proceedings of Mensch und Computer 2019, Hamburg, Germany, September 8-11, 2019*, pages 885–888, 2019.

[YHR$^+$19] Enes Yigitbas, André Hottung, Sebastian Mansfield Rojas, Anthony Anjorin, Stefan Sauer, and Gregor Engels. Context- and data-driven satisfaction analysis of user interface adaptations based on instant user feedback. *PACMHCI*, 3:19:1–19:20, 2019.

[YJJ$^+$19a] Enes Yigitbas, Klementina Josifovska, Ivan Jovanovikj, Ferhat Kalinci, Anthony Anjorin, and Gregor Engels. Component-based development of adaptive user interfaces. In *Proceedings of the ACM SIGCHI Symposium on*

Engineering Interactive Computing Systems, EICS 2019, Valencia, Spain, June 18-21, 2019, pages 13:1–13:7, 2019.

[YJJ+19b] Enes Yigitbas, Ivan Jovanovikj, Klementina Josifovska, Stefan Sauer, and Gregor Engels. On-the-fly usability evaluation of mobile adaptive UIs through instant user feedback. In *Proceedings of the 17th IFIP TC.13 International Conference on Human-Computer Interaction (INTERACT 2019) (to appear)*, 2019.

[YJSE19a] Enes Yigitbas, Ivan Jovanovikj, Stefan Sauer, and Gregor Engels. A model-based framework for context-aware augmented reality applications. In *Handling Security, Usability, User Experience and Reliability in User-Centered Development Processes (IFIP WG 13.2 and WG 13.5 International Workshop at INTERACT2019)*, 2019.

[YJSE19b] Enes Yigitbas, Ivan Jovanovikj, Stefan Sauer, and Gregor Engels. Towards model-based development of context-aware augmented reality applications. *Softwaretechnik-Trends, Proceedings of the 21st Workshop Software-Reengineering and Evolution (WSRE) and 10th Workshop Design for Future (DFF)*, 39(2):39–40, 2019.

[YKUS16] Enes Yigitbas, Thomas Kern, Patrick Urban, and Stefan Sauer. Multi-device UI development for task-continuous cross-channel web applications. In *Current Trends in Web Engineering - ICWE 2016 International Workshops, DUI, TELERISE, SoWeMine, and Liquid Web, Lugano, Switzerland, June 6-9, 2016, Revised Selected Papers*, pages 114–127, 2016.

[YMS15] Enes Yigitbas, Bastian Mohrmann, and Stefan Sauer. Model-driven UI development integrating HCI patterns. In *Proceedings of the 1st Workshop on Large-scale and Model-based Interactive Systems: Approaches and Challenges, LMIS 2015, co-located with 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2015), Duisburg, Germany, June 23, 2015.*, pages 42–46, 2015.

[YS14] Enes Yigitbas and Stefan Sauer. Flexible & adaptive UIs for self-service systems. In *Mensch & Computer 2014 - Workshopband, 14. Fachübergreifende Konferenz für Interaktive und Kooperative Medien - Interaktiv unterwegs - Freiräume gestalten, 31. August - 3. September 2014, München, Germany*, pages 167–175, 2014.

[YS16a] Enes Yigitbas and Stefan Sauer. Customized UI development through context-sensitive GUI patterns. In *Mensch und Computer 2016 - Workshopband, Aachen, Germany, September 4-7, 2016*, 2016.

[YS16b] Enes Yigitbas and Stefan Sauer. Engineering context-adaptive UIs for task-continuous cross-channel applications. In *Human-Centered and Error-Resilient Systems Development - IFIP WG 13.2/13.5 Joint Working Conference 6th International Conference on Human-Centered Software Engineering, HCSE 2016, and 8th International Conference on Human Error, Safety, and System Development, HESSD 2016 Stockholm, Sweden, August 29-31, 2016, Proceedings*, pages 281–300, 2016.

[YSE15] Enes Yigitbas, Stefan Sauer, and Gregor Engels. A model-based framework for multi-adaptive migratory user interfaces. In *Human-Computer Interaction: Interaction Technologies - 17th International Conference, HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015, Proceedings, Part II*, pages 563–572, 2015.

[YSE17] Enes Yigitbas, Stefan Sauer, and Gregor Engels. Adapt-UI: An IDE supporting model-driven development of self-adaptive UIs. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2017, Lisbon, Portugal, June 26-29, 2017*, pages 99–104, 2017.

[YSSE17] Enes Yigitbas, Hagen Stahl, Stefan Sauer, and Gregor Engels. Self-adaptive UIs: Integrated model-driven development of UIs and their adaptations. In *Modelling Foundations and Applications - 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, pages 126–141, 2017.

[zA97] Pinar Öztürk and Agnar Aamodt. Towards a model of context for case-based diagnostic problem solving. In *in Context-97; Proceedings of the interdisciplinary conference on modeling and using context, (Rio de Janeiro*, pages 198–208, 1997.