FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Application of User Interaction Description Languages to provide accessible interfaces for elderly people

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Andreas Kuntner
Matrikelnummer 0426716

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Assoc. Prof. Dipl. Ing^{in} Dr^{in} Hilda Tellioglu

Wien, 14.10.2012

(Unterschrift Verfasserin) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Andreas Kuntner
Wurmsergasse 43/15, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Abstract

This thesis suggests the enhancement of Ambient Assistant Living (AAL) environments by applying concepts from the research field of User Interaction Description Languages (UIDL). In detail, it should evaluate how user interfaces for interacting with AAL services can be generated in an automatic or semi-automatic way based on generic user interaction descriptions, and how the users of AAL environments can benefit from such an approach.

This is accomplished by defining a list of criteria for evaluating the applicability of existing User Interaction Description Languages to the field of Ambient Assistend Living. Existing solutions were analyzed and compared with each other based on these criteria as part of this thesis.

The class of mobile devices, including both smartphones and tablet devices, was defined as exemplary target platform for evaluating the possibilities of generated user interfaces. Interviews and user workshops were conducted using mockups of potential user interactions that might occur in common AAL services, in order to analyze how the typical user group of AAL environments interacts with devices belonging to this class. Guidelines for the implementation of a user interface generation system were established based on the results of this investigation.

In addition, a prototype version of a user interface generation system for use in AAL environments was developed, including two exemplary services and a front-end solution for mobile devices.

For evaluating this software product, it was formally analyzed based on the criteria mentioned above, and compared to the other mentioned UIDL solutions. In addition, practical tests were performed in cooperation with members of the system's prospective user group in order to evaluate how well those potential users can interact with AAL services using automatically generated user interfaces.

Results prove that it is possible to define the user interactions offered by AAL services in a detailed, but nevertheless completely modality- and implementation-independent way, and to create attractive, functional, and accessible user interfaces in a fully automatic way based on these abstract interaction definitions. The evaluation carried out shows that potential users approve the idea of executing AAL services from a variety of devices they are accustomed to. However, the target platform of smartphone and tablet devices proved not to be the ideal choice for this user group, since group members lack experience with the general handling of those devices.

Kurzfassung

Die vorliegende Diplomarbeit behandelt Konzepte aus dem Forschungsgebiet der User Interaction Description Languages (UIDL) und deren Anwendbarkeit auf Ambient Assisted Living (AAL) Umgebungen. Konkret soll die Frage untersucht werden, auf welche Weise User Interfaces, die die Interaktion mit AAL-Systemen erlauben, automatisch oder halbautomatisch generiert werden können, und welche Vorteile die BenutzerInnen von AAL-Systemen davon haben.

Dazu definiert die Arbeit einerseits einen Katalog von Kriterien, anhand derer die Anwendbarkeit existierender UIDL-Lösungen in AAL-Umgebungen beurteilt werden kann. Mehrere solcher Lösungen wurden anhand dieser Kriterien analysiert und miteinander verglichen.

Als beispielhafte Referenzplattform wurden mobile Geräte ausgewählt, diese Geräteklasse umfasst sowohl Smartphones als auch Tablets. Anhand von typischen Benutzerinteraktionen die in AAL-Diensten vorkommen können, wurde im Rahmen von Interviews und Workshops überprüft, wie potentielle NutzerInnen von AAL-Systemen mit Geräten aus dieser Referenzklasse interagieren. Basierend auf den Ergebnissen dieser Untersuchung wurden Richtlinien erstellt, die bei der Entwicklung eines Systems zur Generierung von User Interfaces helfen sollen.

Im Rahmen der Arbeit wurde weiters ein Prototyp eines solchen Systems implementiert, der User Interfaces für mobile Geräte basierend auf generischen Beschreibungen der möglichen Benutzerinteraktionen automatisch generiert. Zwei beispielhafte AAL-Dienste wurden definiert und ebenfalls implementiert, und sind Teil dieses Prototyps.

Abschließend wurde die oben genannte erste Version des Systems basierend auf den im vorderen Teil definierten Kriterien analysiert und mit existierenden UIDL-Lösungen verglichen. Um die Praxistauglichkeit des Systems zu beurteilen, wurden praktische Tests mit Personen aus der typischen Benutzergruppe von AAL-Umgebungen durchgeführt.

Die in der Arbeit präsentierten Ergebnisse beweisen, dass es technisch möglich ist, vollautomatisch ansprechende User Interfaces, die von BenutzerInnen auch gut bedient werden können, zu generieren. Weiters zeigt sich, dass potentielle BenutzerInnen von AAL-Umgebunden die Idee, beliebige Geräte zur Steuerung von AAL-Diensten verwenden zu können, grundsätzlich befürworten. Allerdings ist die gewählte Beispielplattform der mobilen Geräte nicht passend für die an den praktischen Tests teilnehmende Benutzergruppe, da in dieser Gruppe kaum Erfahrung mit der Bedienung von Smartphones und Tablets vorhanden ist.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation and problem statement

During the last years, electronic consumer devices have become more and more important in our lives. Nowadays, the handling and use of various types of such devices in every-day life is considered an integral part of our daily routine. This does not primarily aim at the use of personal desktop computers for accomplishing daily tasks. In fact, the classic type of personal computers is more and more often supported - and in many fields even replaced - by new types of electronic devices which are embedded into every-day life. Tasks that were restricted to sitting at the desk using a classic PC a few years ago can nowadays be performed while lying on the sofa using a television set, or while being on the way using a smartphone device.

Apart from flexibility, convenience and integration into every-day life routines, another reason for electronic devices such as laptop computers, smartphones, tablet devices, etc. being so popular is interconnectedness: Nowadays many typical consumer devices are connected to the Internet, thus enabling access to personal documents as well as publicly available information in virtually all imaginable situations and environments.

One obvious implication is the connection of complex electronic systems with such electronic consumer devices which have become everyday items for most users. This allows, for example, the relatively easy operation of complex networks and systems through consumer devices. This is a promising approach particularly when applied in the field of Ambient Assisted Living environments. Especially the user group of elderly people could benefit from the application of everyday devices they are used to as alternative to complex devices that require training and special instructions.

The term *Ambient Assisted Living* (AAL) includes systems and services that aim at improving and facilitating every-day life of elderly and handicapped persons, and reducing the necessity of external support and assistance for these persons. AAL environments may contain various sensors, devices, and services which act in the background, allowing elderly and handicapped persons to live in the environment they are accustomed to as alternative to moving to a nursing home. Patients are supported in their daily routine by AAL services executed by the AAL environment. AAL services may be executed automatically, for example services that use sensors to detect and react to cases of emergency, or manually, for example services that are intended to assist patients in accomplishing certain tasks. The approach discussed in this thesis focuses on the latter, AAL services that are manually started by patients on purpose in order to achieve a certain task.

This type of AAL services usually requires interaction between the underlying system coordinating the various components of an AAL environment and the user, in this case the patient, after being started. This includes both information that is to be conveyed from the AAL service to the user, and data required by the AAL service that needs to be input by the user. The idea of using electronic consumer devices as an interface between patients and AAL environment brings about the following advantages compared to an AAL environment that provides interaction with users only through desktop computers or specialized devices:

- Through keeping the amount of training and preparation time necessary for controlling an AAL system low, consumer acceptance for AAL environments may be increased. This

applies especially to the user group of elderly people which lack experience in the handling of personal computers compared to young persons who have been used to handle PCs through most of their lives.

- Especially physically impaired persons might benefit from the possibility to choose any device that compensates their handicap for controlling a system that is intended to assist them.

This requires the integration of as many types of devices to be used for controlling AAL services as possible. Only AAL environments that allow users to choose from a maximum number of devices can realize the two advantages mentioned above in the best way. A crucial factor is the support of devices featuring different input and output modalities, in order assist even users with physical impairments. The term *modality* regards the exact way of presenting data to the user, or requesting data input from the user: Different types of devices might address different human senses. For example, information to be conveyed to the user may be displayed on a graphical screen, but it may also be output using voice through a loudspeaker, or presented using tactile signals.

One way of integrating a maximum number of different devices and ensuring interoperability between all those devices and AAL services is the usage of generic user interactions. Using this approach, an application such as an AAL service specifies all potential user interactions it supports, instead of explicitly defining the exact layout of its user interface. Concrete user interfaces for different types of devices can then be created, either automatically or semi-automatically, based on this description of user interactions.

Since explicitly defined user interfaces are bound to a specific toolkit and programming language or markup language, for each class of device to be supported a separate user interface definition has to be created manually. Compared to this approach of manual user interface creation, the automatic or semi-automatic generation of user interfaces based on interaction descriptions considerably reduces the manual effort necessary for integrating a large number of devices and therefore automatically increases the number of supported devices.

The major challenge of using such an approach of automatic or semi-automatic user interface generation is the quality of the rendered user interfaces. Especially when developing for a user group mainly consisting of elderly people who might not be experienced in the use of electronic devices and handicapped persons who are limited to certain input and output modalities, it is important to provide products that ensure maximum accessibility. The term *accessibility* in this context refers to the ability of each potential user to access all functions of a device or a software product in the way that suites his personal and physical conditions best.

1.2 Research question and methodology

The objective of this thesis is to answer the question how user interfaces for different modalities can be generated in an automatic or semi-automatic way based on generic user interaction descriptions, and how the users of those rendered interfaces can benefit from such an approach.

To achieve this, existing solutions for the definition of generic user interactions and the generation of user interfaces - subsumed as *User Interaction Description Languages* (UIDL) - are examined and compared to each other in chapter 2. A list of criteria for analyzing existing solutions is established, and the central concepts employed by these solutions are analyzed and compared in order to examine which of these concepts are best suited for being integrated in a user interface generation system developed especially for use in AAL environments.

In addition, a concrete result of this thesis is the practical implementation of a prototype of such a user interface generation system. This system aims specifically at AAL environments and the generation of accessible user interfaces for elderly people, focusing on one exemplary type of end user device. This prototype is based on the results of the comparison of the various concepts used by existing User Interaction Description Languages and on interaction mechanisms developed in cooperation with potential users of the system.

The special needs of the prototype's user group, especially in relation to accessibility, require the continuous involvement of persons belonging to the user group in the whole design process. Therefore workshops were performed in cooperation with potential users prior to implementing the prototype in order to establish design guidelines for the implementation. One outcome of this thesis, in addition to the prototype implementation, is the exact documentation of these user workshops. This documentation is contained in chapter 3, while details concerning the implementation of the prototype are discussed in chapter 4.

For evaluating the prototype of the user interface generation system, two exemplary AAL services are defined as part of this thesis. The prototype is compared to the existing User Interaction Description Languages that were analyzed, and evaluated using practical user tests with persons belonging to the target user group of the system, in chapter 5.

Finally, chapter 6 summarizes the results and outcomes of this thesis. In addition, some open research topics that may offer interesting research work in the future are identified.

CHAPTER 2

State of the art

Parts of the contents of this chapter have been published in:

> Report on user interface analysis. *Deliverable D-3.1 of the AAL Joint Programme "Ambient Assisted Living user interfaces" (AAL-2010-3-070)*, 2011. http://www.aaluis.eu/wp-content/uploads/2012/05/D3.1-Report-on-User-Interface-Analysis.pdf

Section 3 "User Interface Description Languages" of Deliverable D-3.1 is based on the research work done by Andreas Kuntner that is presented in the following chapter of this thesis.

2.1 Comparison criteria

This section provides a list of seven criteria that can be used to analyse User Interaction Description Languages (UIDL) and identify their applicability to Ambient Assisted Living environments. In addition, they can be used for comparison of multiple UIDLs. In the following sections, several UIDLs are presented in detail and analyzed based on these criteria, allowing easy comparison.

Items 3 to 6 of the following list are use-case independent, as they concern general quality criteria that might be of interest for the analysis of any markup language, as suggested by [21]. For example, when deciding which language to choose for practical use, it is quite important to analyze if the full specification is available for use, and whether the language is still under development or not.

Items 1 and 7 concern the abstraction of a certain UIDL as they analyze how generic it is and to how many cases it can be applied. Finally, item 2 focuses on AAL environments, as it analyses characteristics that are important especially to the typical user group of such systems.

In summary, all three types of characteristics are important in order to identify those languages that are both widely used and best suited for AAL environments.

1. **Level of abstraction**

   UIDLs must be sufficiently abstract to allow the creation of multimodal user interfaces, meaning user interfaces for different devices that use different interaction modalities. For example, the user interaction with a PC with mouse and keyboard is totally different from that with a mobile phone with multitouchscreen, or a car radio featuring only voice interaction.

   Requirements for a high level of abstraction are that no information about the use of specific user interface widgets is encoded, as well as that no concrete layout information is given since this information might only be used by graphical user interfaces on a certain type of display.

2. **Adaptability**

   This criterion concerns the possibility to adapt user interfaces automatically based on different environmental settings. For AAL environments, the following three characteristics are important:

   - Accessibility: The user interface should automatically adapt to user preferences, based on a user's abilities and disabilities. This is important especially to users of AAL environments, because most of them use these systems to overcome physical disabilities.
   - Use-case awareness: In different use cases, different user interface (UI) devices are used. For example, for activating and deactivating a service, a mobile phone is mostly used because the user can carry it with him or her, while changing the basic setup of a service is carried out using a PC due to the more sophisticated input mechanisms.

UIDLs should know about the capabilities of UI devices and automatically adapt user interfaces to provide different functionalities on different devices as well as present the user interface in the way that is supported best by each type of device.

- Context-awareness: Finally, it is desirable to automatically adapt the presentation of a user interface based on environmental influences, for example physical conditions such as the intensity of light around the UI device.

3. **Openness**

   This criterion evaluates how much information about a UIDL is freely available, and how detailed this information is. In addition, it concerns licensing issues.

4. **Organizational background**

   This criterion states which type of organization started the development of a UIDL, and if it has an industrial or a research background. Examples for organizational types are company, university, research organization, open-source-community, etc.

5. **Status**

   The aim of this criterion is to give an overview of the history and the current development status of a UIDL. It answers the following questions:

   - When did the development of the first version start?
   - How many versions have been published?
   - Which is the latest version, and when has it been published?
   - Is the language still further developed?
   - Is a standardization process of the language specification planned, or has it been accepted as national / international standard already?

6. **Number of implementations** In general, an overview of all pieces of software that make use of a UIDL in some way might be interesting since it indicates how common the language is and how it is used in practice. For analyzing the applicability of a UIDL to different usage scenarios, the number of target toolkits supported by the language is of special interest. As most systems require external libraries to provide information about how to generate a toolkit-specific user interface, this criterion would correspond to the number of available libraries for different target toolkits.

7. **Number of supported target platforms**

   This criterion is related to the previous one, although the number of target platforms supported by a UIDL may differ from the number of supported target toolkits. Multiple user interface toolkits may target the same platform, for example there are several toolkits available for UI presentation on PCs. At the same time, a UI toolkit may support multiple platforms. In this case, user interface generation information is needed separately for each platform.

2.2 User Interaction Description Languages

This section presents and analyzes 15 existing User Interaction Description Languages. For each of these, the detailed syntax and language specification is summarized and its applicability to AAL environments is evaluated based on the criteria established in the previous section.

2.2.1 Alternate Abstract Interface Markup Language (AAIML)

In the beginning of the 2000s, the V2 technical committee of the International Committee for Information Technology Standards (INCITS), an international standardization organization, started an initiative to develop an abstract interface description language. This initiative was part of the Universal Remote Console (URC) specification that is described in detail in section 2.2.2.

The authors of [47] present the first draft of the URC specification, part of which is the Alternate Abstract Interface Markup Language (AAIML). This is a markup language used for defining UI descriptions in an abstract way, based on XML. An AAIML model defines a user interface in an abstract way. In practice, an AAIML document would contain an AAIML model that describes the user interface of exactly one target service that should be controlled from a remote console device.

Language specification

An abstract AAIML model is composed by abstract UI elements which are called interactors. Each interactor encodes a specific input or output operation. The set of possible interactors that can be used for composing user interfaces is predefined in the AAIML specification, in order to ensure that interactors are defined in a modality-independent way. This precondition is necessary to guarantee that every user interface defined in AAIML can be rendered on any user input/output device, using any input/output modality the device is capable of.

In addition to abstract interactors, the AAIML specification provides supplementary language elements that allow incorporating meta-information in the UI description. This includes the following:

- Group elements allow to group several interactors that should be presented collectively or next to each other. This can be useful if several interactors provide similar functionality, or if they are all necessary to finish a certain task.

- Help attributes may be attached to the whole user interface, to single interactors, or to options inside an interactor. Moreover a help attribute can contain several help texts, all containing the same information but at different levels of granularity. Therefore it is possible to present a short help text on each UI component, and provide more detailed information on user request.

- Interactors may be assigned to different classes, each class representing a certain level of detail. By assigning interactors to classes, implicitly the importance of the functionality

each interactor represents is encoded. On visual user interface devices that provide limited screen space, for example, only those interactors that represent the most important features could be shown.

Information flow and rendering

While the system is in use, the abstract UI description would be transmitted from the target service to the specific user interface device the user wants to use for controlling the target service device. Every device can be used as user interface device as long as it implements the URC standard. On the user interface device, the abstract AAIML model is transformed to a concrete UI model by an AAIML renderer. The concrete UI model is encoded in any device-specific user interface description language that can be directly rendered by the user interface device.

Unfortunately, no detailed information is available about how the transformation process between abstract model and different concrete UI implementations is accomplished. However, according to [47] several implementations exist that allow transformation to different toolkits using different modalities, among them graphical user interfaces (rendered using Java Swing), speech output using a text-to-speech generator, and Braille-based input/output (using the Windows CE platform).

Status

The AAIML language is not part of the final URC specification, but rather has evolved to the ”presentation template markup language”. The whole URC specification has been adopted as a national standard in the US [3] in 2005, and as an international standard [4] in 2008. It seems that work on the original AAIML language has been discontinued.

AAIML is mentioned only for reasons of completeness, since it is the predecessor of the User Interaction Description Language incorporated in ISO/IEC 24752 and shows the original principles of the first version of the language. The ISO/IEC 24752 standard is presented in detail in section 2.2.2.

Analysis

The following enumeration contains a detailed analysis of the Alternate Abstract Interface Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   In theory, the level of abstraction is quite high due to the use of abstract interactor object that are dynamically transformed to concrete UI elements at runtime. However, no examples for such abstract interactors are freely available, and no detailed information about the transformation process from abstract to concrete UI elements in practice is provided.

2. **Adaptability**

   As part of the abstract user interaction description language, classes may be defined that restrict certain interactions to special devices (and therefore to certain use-cases), therefore use-case awareness is fulfilled by the URC specification. The framework however does not specify how to incorporate user preferences or environmental settings in the transformation process of abstract to concrete user interface elements. Also no information about the availability of interfaces for external specifications is available, thus the degree of adaptability concerning accessibility and context-awareness is unknown.

3. **Openness**

   Not much information is available concerning the AAIML language, and it is unclear when exactly the original AAIML language draft was superseded by the presentation template markup language incorporated in the final URC specification.

4. **Organizational background**

   The approach was proposed and implemented by the INCITS standardization organization.

5. **Status**

   The User Interaction Description Language originally known as AAIML has evolved into the URC specification, the work on the original version of the language is currently discontinued.

6. **Number of implementations**

   In 2002, five prototypical implementations were available, and two more were under development. Unfortunately, no information about the further development of these two additional prototypes can be found.

7. **Number of supported target platforms**

   The five prototypes that were available in 2002 aimed at five different target platforms: Web browsers (to be viewed on a PC), handheld devices, voice-based input and output systems, Braille devices, and augmentative and alternative communication systems.

2.2.2 Presentation Template (PreT) markup language (ISO/IEC 24752)

The ISO/IEC 24752 standard [4] is the direct successor of the ANSI/INCITS 389 through 393 standards family. This international standard titled ”Information technology - User interfaces - Universal remote console” defines the latest specification of the Universal Remote Console (URC) specification. The basic setup of a URC environment consists of one or multiple target devices that provide services the user can access through the use of one or multiple user

level of abstraction:	high
adaptability:	
accessibility:	unknown
context-awareness:	unknown
use-case awareness:	yes
openness:	low
organizational background:	standardization organization
status:	discontinued
implementations:	5
supported target platforms:	5

Table 2.1: Comparison chart for the Alternate Abstract Interface Markup Language (AAIML)

interface devices. The concrete user interface is rendered individually for the UI device's platform at runtime, based on the capabilities of the available services, thus in theory supporting an unlimited number of UI devices.

Compared to conventional remote control systems, this approach has the following advantages:

- Target devices that are not provided with specialized remote controls by their manufacturer may be used more conveniently when they can be controlled through a remote device.
- One user interface device can be used to control all target systems (traditionally, each target device ships with its own remote control).
- The user can freely choose which user interface device to use as remote control for all these devices.
- A user interface device may not only function as remote control, but also as remote console. Traditional remote controls allow the user to send commands to the target device, while remote consoles are able to display feedback or information about the target's current state to the user as well.

The capabilities of the target services are specified in target descriptions, their public interfaces to be used by user interface devices are specified in user interface sockets, and abstract user interfaces are specified for each target service in presentation templates. The ISO/IEC 24572 standard consists of five parts, resembling these necessary specifications: The first part describes the general principles of the URC framework. The second, third, and forth part describe how to specify user interface sockets, presentation templates, and target descriptions, respectively. Finally, the fifth part specifies additional resource descriptions.

Language specification

For a detailed insight into the above-mentioned standard's user interaction description language, the Presentation Template (PreT) markup language that is specified in ISO/IEC 24572-2 [6], it is necessary to analyze both the PreT language itself and the user interface socket description language specified in ISO/IEC 24572-3 [5]. Technically, both types of definition are specified using an XML syntax.

User interface sockets

In general, user interface sockets are *"an abstract concept that, when implemented, exposes the functionality and state of a target in a machine-interpretable manner"* [6]. A user interface socket contains all data that is internally present in a target service implementation and should be made accessible to the user. This includes (public) variables and functions as well as notifications. User interface sockets are usually platform-independent, although they allow the specification of platform-specific mapping information in addition to any objects, in which case the user interface socket loses its platform neutrality.

For referencing variables, the user interface socket specification defines objects of type `variable` and `constant`. As is convention in most programming languages, variables are used for data that changes throughout the execution of the service, while constants must be defined before runtime and stay fixed. Both types, however, are restricted to data that is presented to or (in case of a variable) edited by the user at any point in the execution of the service. Variables or constants that are used internally by the service to store data temporarily, for example, are not represented in a user interface socket description at all, because user interface sockets serve only as interface between service and user interface. Internal data need not to be represented in the user interface.

Objects of type `command` represent an interface to public functions of the target service. Through a command, the user can directly call a function of the service that accomplishes a certain task and can not be represented as a variable. Service-internal functions that simply change the value of a variable are not represented as a command, in such cases the variable itself would rather be represented in the user interface socket description.

In addition, user interface sockets allow the definition of `notifications`. In general, the purpose of a notification is to inform the user about the current state of the system. A notification is always triggered by the target service (although it can be triggered as feedback to incorrect input received from the user). When a notification is triggered by the service, the normal operation of the service is suspended.

A user interface socket description starts with global declarations, such as ID, date of the last modification, etc. Furthermore it contains a set of objects of type variable, constant, command, or notification. In addition, also the definition of objects of type `set` is allowed. Sets are used to group other objects and to express hierarchical structure, since they may be nested.

To allow specifications to be as detailed as possible, each type of object has additional attributes and sub-elements, some of them optional and some mandatory. For example, variables and constants must have a `type` attribute defining the data type of the underlying variable. In

addition to simple types as defined in XML Schema Part 2 [27] and enumerations, custom types may be defined in a separate section of the user interface socket definition. The `dependency` sub-element allows to further specify the minimum and maximum size of a variable, a regular expression pattern that the content of a string variable must match, a flag indicating whether the user may change the content of a variable, an expression of how to calculate the content of a read-only variable from several other variables, and similar attributes. All those dependencies may be specified as XPath [19] expressions that evaluate to simple data types at runtime.

Commands may define any number of `param` sub-elements, each representing one input or output parameter passed to the method that the command represents. A param attribute must reference a variable and define its direction, which can either be input or output. Similar to variables, also commands may have a `dependency` sub-element that contains, for example, a flag that indicates whether the user may activate the command at a certain point of time in the execution of the service.

Notifications require the specification of a `category` attribute that equals to either ”info”, ”alert”, or ”error”, giving hints to the user interface device on how the notification shall be presented to the user. In addition, notifications provide attributes that indicate if the user has to explicitly acknowledge the notification or if it times out after a certain period of time.

All objects present in a user interface socket definition may provide an optional `mapping` sub-element. This element must have a `platform` attribute that specifies a certain target platform. The content of a mapping element are not restricted by the user interface socket specification, thus allowing any XML-based sub-elements. The purpose of the mapping element is to encode platform-specific mapping information for the mapping of the mapping element's parent object to a certain target platform.

Presentation Templates

A Presentation Templates (PreT) as defined in the presentation template markup language maps elements of a user interface socket to interaction mechanisms [5]. It contains abstract interactors that are used for either presenting data to the user, or requesting input data from the user. Each of these interactors is bound to exactly one element of the corresponding user interface socket definition. All interactors are sufficiently abstract to ensure that presentation templates can be used in any delivery context, meaning they are modality-independent.

A presentation template contains global declarations such as ID, date of the last modification, etc. and a set of interactor objects. Interactors are contained in `group` elements, allowing to group interactors with similar semantics and express hierarchical relations between interactors, since group elements may be nested. To express a distinct order in which interactors shall be presented to the user, all interactor elements as well as the group element can be assigned a number via the `navindex` attribute. There exist no interactor attributes for labels, help texts, keywords, etc. since those elements shall be defined as external resources using the resource description markup language specified in ISO/IEC 24572-5 as part of the URC specification. Resources are mapped to the corresponding PreT interactors by referencing presentation templates and interactors in the resource specification.

Interactors specified by the PreT markup language can be distinguished by their category:

- **Input interactors**

  There are several interactors that represent a request for user input. The simplest is the `input` interactor which can be bound to variables defined in a user interface socket and allows free-form data entry to be stored in that variable. For input of data in a way that is not monitorable for any third party present, the `secret` interactor can be used. The `textarea` interactor is intended for input of multi-line text-based content.

  Both the `select1` and the `select` interactors allow selections from a list of multiple choices, the former restricting the selection to exactly one choice, the latter allowing any number of choices. Finally, the `range` interactor allows the selection from a sequential range of values.

- **Output interactor**

  The `output` interactor is used to present data of any type to the user which may not be edited. An output interactor can be bound to a variable or a constant defined in a user interface socket, but also to a command which is useful for displaying the state of the command.

- **Command interactor**

  The `trigger` interactor is bound to a command defined in a user interface socket and can be used by the user to activate that command.

- **Notification interactor**

  The `modalDialog` interactor is a special interactor that suspends the normal user interaction to present information, warnings or errors. It can be used as representation of a notification element defined in a user interface socket. This type of interactor is the only one that may not be included in a `group` element but rather be defined globally, as direct sub-element of the presentation template. The `modalDialog` interactor may contain other interactors, for example `output` interactors can be used to display detailed information concerning the current notification, and `trigger` interactors can offer the user a possibility to acknowledge the notification.

Information flow and rendering

When using a URC-compliant user interface device to control a URC-compliant target service, the UI device would request the presentation template from the device and directly render and display a user interface based on the PreT definition. In practice, however, this architecture is restricted to a very small number of devices, since the manufacturer of each end device (including both service and user interface devices) needs to ensure that the device implements the full URC specification.

As an alternative, the Universal Control Hub (UCH) architecture was implemented [46]. The UCH architecture is an extension to the URC standard, which introduces a URC-based gateway between target services and user interface devices, allowing end devices that are not URC-compliant to communicate as proposed in the URC specification. This approach therefore

circumvents the restriction that direct communication is only possible when all target devices and all UI devices implement the URC specification.

Using UCH, the gateway transforms the presentation template delivered by the service to a concrete user interface, written in any programming or user interface description language that the user interface device's platform can interpret. The only precondition for this transformation process is that the gateway knows how to map abstract interactors to elements in the target language, or that an external source of information is available that describes this mapping. The concrete user interface description is then delivered to and renderer and displayed by the UI device. Similarly, when the target service delivers data to be presented to the user, or requests data from the user, the data is first processed by the gateway and transformed to formats readable by the respective device.

Status

The current URC specification is the successor of the ANSI/INCITS 389-393 [3] family of standards and has been adopted as international standard by the International Organization for Standardization (ISO) in 2008. The standard consists of the following five parts:

1. Framework (ISO/IEC 24572-1)
2. User interface socket description (ISO/IEC 24572-2)
3. Presentation template (ISO/IEC 24572-3)
4. Target description (ISO/IEC 24572-4)
5. Resource description (ISO/IEC 24572-5)

After the publication of the ANSI/INCITS 389-393 standards in 2005, the URC consortium[1] was founded in order to promote and implement the URC standards and prepare the submission for an ISO/IEC standard. The consortium consists of individual members of research organizations, universities and industrial organizations from Europe, India and the US. Several implementations of the URC standard have been developed, among them the Universal Control Hub (UCH) Reference Implementations.

Analysis

The following enumeration contains a detailed analysis of the Presentation Template markup language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   The level of abstraction is quite high, although there are some restricting factors concerning both user interface sockets and presentation templates:

[1] `http://myurc.org/urcc.php`

- Theoretically, presentation templates are modality-independent as the interactors they consist of are sufficiently abstract to be used in any delivery context. In practice, however, the set of proposed interactors seems to be quite closely related to graphical user interfaces. One example concerns the `input` and `textarea` interactors: To distinguish between short and long string data types may be useful in several delivery contexts, however the above-mentioned interactors use the number of lines to express this distinction (single-line text input is accomplished by using the `input` interactor, multi-line text input is accomplished by using the `textarea` interactor). When using voice-based interaction, for example, this criterion is not available since spoken text is not divided into lines, therefore in such an environment there is no difference between the `input` and the `textarea` interactor.
- There is no guarantee that user interface sockets are platform-independent, due to the potential use of `mapping` elements in the socket definition.

2. **Adaptability**

   The main goal of the URC standard is to let the user choose which device to use as remote console for different target devices, which is an important step towards accessibility (especially when compared to traditional systems that force the user to use a specific remote control). However, an important improvement would be the option of defining preferences on how user interfaces are are a specific device (for example, using a high contrast on graphical UI devices, or a certain level of loudness on voice-based interaction systems). The URC specification does neither incorporate a user preference definition system, nor offer interfaces for external user preference descriptions.

   Adaptability concerning different use-cases or contexts of use is currently not supported by the URC framework.

3. **Openness**

   The URC Standards Whitepaper [44] provides a basic overview of the elements of the first URC specification that formed the ANSI/INCITS 389-393 family of standards, but detailed information on the current (2008) version is only available from the ISO/IEC 24572 standard documents which are not freely available.

4. **Organizational background**

   The approach was proposed by the INCITS standardization organization in the US, approved by the ISO international standardization organization, and is currently promoted and implemented by an international consortium containing research organizations, universities and industrial businesses.

5. **Status**

   The URC specification was approved as a US national standard in 2005 and as international standard in 2008, implementational work is currently in progress by the URC consortium.

6. **Number of implementations**

   A reference implementation of the UCH architecture that is based on the URC framework is available for the Java and C/C++ programming languages, developed by the University of Wisconsin-Madison. Based on this reference implementation, several prototypical tools were developed[2]:

   - a HTML-based user interface client for the UCH architecture for iOS operating systems (iPhone etc.)
   - a user interface generator that creates interface templates based on HTML and JavaScript automatically, both for direct deployment to user interface devices and for refinement by UI designers
   - a Flash-based user interface client for the UCH architecture for mobile devices (PDAs)
   - a Web 2.0 user interface for use in home media environments that is based on the UCH architecture
   - a task-based user interface client for mobile devices (PDAs) for controlling media servers running the UCH architecture

   Moreover, there is at least one research project making use of the UCH framework: The EU-funded i2home project aimed at developing an open reference architecture for intuitive interaction of elderly and cognitively disabled persons with home appliances [8].

7. **Number of supported target platforms**

   The prototypes developed by the URC consortium aim at use with mobile devices and desktop computers.

level of abstraction:	rather high
adaptability:	
accessibility:	medium
context-awareness:	no
use-case awareness:	no
openness:	medium
organizational background:	international consortium
status:	approved as international standard
implementations:	> 5
supported target platforms:	2

Table 2.2: Comparison chart for the Presentation Template markup language

[2] `http://myurc.org/tools/`

2.2.3 Extensible Interface Markup Language (XIML)

The Extensible Interface Markup Language (XIML) is a framework for defining abstract user interfaces and transforming them to concrete ones. In addition, it offers several methods for the automatic adaptation of user interfaces, based on different criteria. Its main focus is the use case of a generic user interface, defined once for an application, that can be executed on a variety of devices, using different platforms [39].

Language specification

XIML is an XML-based markup language. It specifies five basic components, that together build the core of an XIML user interface description: User, task, domain, presentation, and dialog components. Task components provide an abstract definition of user interactions, while presentation and dialog components represent concrete user interfaces. Domain and user components contain supplemental information needed for the generation of user interfaces.

- **User component**

  The user component defines all users that may operate the system. The items of the user component are structured in a hierarchical tree, each node in this tree representing either a single user or a group of users. For each of the users or user groups, characteristics can be defined as attribute-value pairs.

- **Task component**

  The task component contains all those parts of the business logic provided by the background service that require interaction with the user. In detail, a task component defines all tasks that the user may wish to accomplish through the user interface the XIML document describes. Those tasks are structured in a hierarchical manner, similar to the structure of a user component. The task component is also responsible for the application flow, meaning that it maintains the correct order in which tasks and interactions are executed.

- **Domain component**

  The domain component contains a collection of all data objects that the user can view or manipulate in the course of any task defined in the task component. The content of a domain component is hierarchically structured, resembling a simple ontology. Objects are stored as key-value pairs.

- **Dialog component**

  The dialog component is the equivalent to the task component on the level of concrete user interfaces. It contains a structured set of interaction items, each specifying in detail one of the interactions that are available to the users of a user interface.

- **Presentation component**

  The presentation component contains all concrete user interface elements that are available in one concrete, platform-dependent user interface. Each concrete user interface element

references a UI widget present in a particular widget toolkit. Therefore an XIML interface definition contains several presentation components, one for each target device that shall be supported.

The different components present in an XIML interface definition are related to each other in different ways. XIML does not provide a predefined, closed set of relations but allows the definition of custom relations. These relations can then be used to model connections between elements defined inside the same component as well as elements of different components. For example, relations can be used to state that certain presentation widgets (particular elements of presentation components) can be used to display a certain data type (a particular element of a data component).

Information flow and rendering

The component-driven architecture of every XIML interface definition guarantees a strict separation of business logic, interaction definitions, and the rendering of concrete user interfaces. The business logic should be at all defined beyond XIML's scope, only those parts that require user interaction are directly referenced by XIML, but only in task components. The definition of interactions is limited to dialog and presentation components, and the actual rendering is left to the specific end devices.

As mentioned above, one presentation component is expected for each target platform. Since concrete, toolkit-dependent widgets are referenced by presentation components, user interfaces can directly be rendered on end devices that are capable of parsing XML documents. For other devices, converters are used that help transforming XIML interface definitions to specific target languages.

The creation of specific presentation components for each target platform to be supported, however, involves a lot of manual work and effort. To overcome this problem, the abstract XIML architecture allows following a more generic approach: Instead of using toolkit-dependent widgets, developers may define custom elements as content of presentation components. These custom components would then be mapped to concrete UI widgets by using XIML relations. Presentation components that consist of such custom elements are called intermediate presentation components, because their elements require one additional transformation step before being rendered.

Such a set of predefined relations between intermediate presentation elements and concrete UI widgets can be used in several XIML projects, thus reducing the effort extensively since only one presentation component needs to be created manually per project, instead of one presentation component for each target platform. Relations can even be generated and inserted automatically by an intelligent extension system to XIML, allowing to replace a certain intermediate presentation element by different concrete widgets, depending contextual settings, device capabilities, user preferences, etc.

Analysis

The following enumeration contains a detailed analysis of the Extensible Interface Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   XIML's strict separation of business logic, interaction description, and UI rendering is a crucial factor for a high level of abstraction and therefore support for many different target platforms and modalities. Although the core XIML language is modality-extensible rather than modality-independent (as a new presentation component must be created manually for each additional modality to be supported), this drawback can be overcome by using the concept of intermediate presentation components, as explained above.

2. **Adaptability**

   XIML provides several methods for automatic adaptation of user interfaces, including both built-in functionality and interfaces for external solutions. The specification of user profiles and users' preferences is even one of the five core components of the language, namely the user component. Personalization is integrated into the basic XIML language, meaning the automatic exchange of user interface widgets that present the same information in different ways based on user preferences. Finally, the mechanism of automatically generating rules for mapping intermediate presentation elements to concrete UI widgets provides a flexible way to react to contextual settings, device capabilities, use-cases, user preferences, etc.

3. **Openness**

   An "XIML Starter Kit", including the full XIML language specification, documentation, samples, and tools is available for free after registration from the developer team's web page[3].

4. **Organizational background**

   XIML was developed by the RedWhale Software Corporation[4], a software development company specialized on user interface design concepts.

5. **Status**

   Version 1.0 is the latest version, it was published in 2002. The current development status is unknown.

6. **Number of implementations**

   The "XIML Starter Kit" contains two converters: one for HTML and one for WML (Wireless Markup Language) output. Wireless Markup Language was developed as part of the Wireless Application Protocol (WAP). It is used for the description of web content pages

[3] `http://www.ximl.org`
[4] `http://www.redwhale.com`

to be displayed on old mobile devices that are not capable of rendering HTML or XHTML content [1].

7. **Number of supported target platforms**

   The HTML converter mentioned above was developed for use with desktop PCs, and the WML converted was intended for use on mobile devices such as PDAs.

level of abstraction:	rather high
adaptability:	
accessibility:	yes
context-awareness:	yes
use-case awareness:	yes
openness:	high
organizational background:	industry
status:	latest version from 2002
implementations:	at least 2
supported target platforms:	2

Table 2.3: Comparison chart for the Extensible Interface Markup Language (XIML)

2.2.4 Extensible Interaction Scenario Language (XISL)

The Extensible Interaction Scenario Language (XISL) was designed with regard to changing Internet usage behavior: In the 1990s, the world wide web was primarily an information medium that was accessed from PCs. In the beginning of the 2000s, this changed in two ways. First, with the ”Web 2.0” movement the web became an interactive medium, in which the user could actively participate. Moreover, an increasing number of devices, e.g. mobile phones, became connected to the Internet. Both developments resulted in new modes of interaction concerning online activities.

XISL is a language for describing online multi-modal interaction systems, using an XML-based syntax. As the name suggests, it describes interaction scenarios rather than concrete user interfaces, thus being applicable to many different interaction modalities. XISL is based on existing open standards such as VoiceXML [33] and SMIL [14], but advances their concepts in a modality-independent approach.

Language specification

The basic purpose of an XISL document is to describe any web-based task a user might wish to accomplish, independent of the terminal that is used for accessing the web. This is done by precisely specifying the control flow between user and system, including both the exact data that is exchanged and the direction in which the data is transmitted (either from the system to the

user, or vice versa). In addition, the exact order in which these data transfers take place must be specified. All this can be encoded as a sequence of interactions between user and system. Finally, those interactions must be specified in a modality-extensible way in order to support any potential end device [23].

An XISL document usually contains one `body` element that contains one or more `dialog` elements. Each `dialog` element groups a set of interactions between user and system. Such interactions are represented by `exchange` elements, several of which may be present inside a `dialog` element. In addition, the `dialog` element supports an attribute that controls the sequence in which the contained interactions are executed, either parallel, sequential, or alternatively (based on additional constraints).

Each interaction consists of a request that is output to the user, a value that is input by the user as feedback to the request, and a final action that decides on the further application flow. Respectively, an `exchange` element contains exactly one `prompt` element (resembling the request), one `operation` element (resembling the user's feedback), and one `action` element.

Each `prompt` element outputs exactly one request to the user, and each `operation` element expects exactly one input from the user. Nonetheless both elements may contain an arbitrary number of `output` and `input` elements, respectively. This is due to the special approach XISL uses for handling multi-modality: For each modality that shall be supported by a particular interaction, an individual `input` or `output` element is used, each equipped with an attribute specifying the type of modality it is intended for. The content of the `input` and `output` elements is not restricted by XISL, as it depends on the respective modality. The advantages and disadvantages of this modality-extensibility based approach compared to modality-independent concepts are discussed in section 2.2.4.

Finally, the `action` element may also include `output` elements to inform the user that the feedback was received and processed, for example. In addition, it contains instructions on which interaction to execute next, depending on user input, application state, etc. Similarly, the `dialog` element may contain `begin` and `end` elements that contain instructions on which interaction to execute as the first one, and after the last one, respectively.

Information flow and rendering

XISL introduces a `dialog manager` that acts as a middleware between web services and user interface devices. The dialog manager ensures that the system is terminal-independent and any end device can be used as terminal to the system. It contains a XISL interpreter that parses XISL documents and is responsible for executing the correct interactions at the right time.

When executing an interaction, the dialog manager sends the content of an `input` or `output` element to the current user interface device. The user interface device need not include an XISL interpreter as there is no XISL code for it to parse. The user interface is rendered there based on the platform-dependent code contained in the `input` or `output` element, and user input is transmitted back to the dialog manager which transforms it, if necessary, to be readable by the web service.

Modality-extensibility vs. modality-independence

In contrast to modality-independent approaches that define interactions in a way that can be interpreted and rendered on different modalities, XISL follows a modality-extensible approach: New `input` or `output` elements are added fot each modality an interaction shall support. This concept has the following advantages and disadvantages:

- Not all potential modalities need to be considered when specifying the interaction description language.

  + Compared to modality-independent approaches that need to be sufficiently abstract to support all possible end devices, rather concrete user interfaces can be specified.

- Each interaction may support different modalities.

  + This brings about additional flexibility, since not all interactions may need to support all end devices.
  - Each XISL document needs to be parsed in detail to know which modalities are supported by the system it describes.
  - XISL documents are difficult to extend in order to support additional modalities, since all rules needed for the new modality must be added in multiple sections inside the XISL document itself (compared to other approaches that can be extended by external libraries, for example).

- The `input` and `output` elements contain code not specified by the XISL core language.

  + They may contain source code that is directly interpretable by a certain end device, therefore no additional interpreters need to be implemented on the end device.
  - In different XISL documents, these elements may contain semantically different code, since there is no common definition of the exact content of these elements.
  - Transmission of data between dialog manager and end device is be difficult, since they may use different toolkits and the data might need conversion or re-formatting.

Analysis

The following enumeration contains a detailed analysis of the Extensible Interaction Scenario Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   The concept of describing web service access as a sequence of interaction scenarios is quite generic as it does not include any modality-specific details. The same applies to the application flow model that groups one input and one output interaction together and provides mechanisms for executing these interaction scenarios in any order, depending

on user input, application state, etc. When it comes to a detailed definition of input and output operations, the concept of the XISL language turns to a very concrete one, relying on separate definitions for each modality. As a consequence, the level of abstraction of XISL must be defined as medium: It offers a rather abstract interaction description that needs to be manually extended to be used in practice.

2. **Adaptability**

   The concrete implementation of input and output operations is left to custom extensions, these are responsible for rendering the user interface and adapting it to users' needs or environmental settings. The XISL core language does not provide any framework for automatic adaptation. The same applies to use-case awareness: The modality-extendable system allows input and output interactions to be presented in quite different ways using different modalities, but the execution of different interaction scenarios depending on the currently used modality is not provided.

3. **Openness**

   The full specification of both the core language and a front end extension is available freely on the developer's web page[5], the latest version of both specifications is available only in Japanese language however.

4. **Organizational background**

   XISL was developed by a research group of the Toyohashi university of technology in Japan.

5. **Status**

   Version 1.0 (which was called "Extensible Interaction Sheets Language") was published in January 2002, the updated version 1.1 was published in April 2003. This is the latest version, the current development status is unknown.

6. **Number of implementations**

   A full specification for PC, mobile phone and PDA platforms exists as extension to the core XISL language, published by the same research group that developed the original XISL language[6]. This extension specifies the content of `input` and `output` elements for these platforms. In addition, a runtime environment was developed for XISL as part of the Galatea Project [38] which provides a speech-based input and output system, featuring animated virtual avatars.

7. **Number of supported target platforms**

   The front end extension mentioned above covers traditional PCs and mobile devices, and the Galatea project features speech-based input and output, totaling in three different target platforms.

[5] `http://www.vox.tutkie.tut.ac.jp/XISL/XISL_Web_Site_E/XislSpecE.html`
[6] `http://www.vox.tutkie.tut.ac.jp/XISL/XISL_Web_Site_E/XislFESpecE.html`

level of abstraction:	medium
adaptability:	
accessibility:	no
context-awareness:	no
use-case awareness:	no
openness:	high
organizational background:	university
status:	latest version from 2003
implementations:	at least 3
supported target platforms:	3

Table 2.4: Extensible Interaction Scenario Language (XISL)

2.2.5 Web Services Description Language (WSDL)

The Web Services Description Language (WSDL) specifies an XML-based language to describe web services based on their functionality. It is an open standard developed by the World Wide Web Consortium (W3C) specified in [17]. WSDL was designed as an addition to the SOAP protocol which is widely used for communication with web services. WSDL is used to define the functionality a web service offers and the interactions to access a certain functionality, while SOAP is used to actually access one of the functions offered by a web service. Although WSDL is often used in combination with SOAP in practice, the language is protocol-independent.

Language specification

The basic structure of a WSDL document is defined in part 1 (”Core Language”) of the WSDL specification. A WSDL document describes a web service based on its functionality, i.e. the operations it offers and the interactions necessary to access these operations. Such a description can be split into two parts:

- The concrete level: The `service` element defines the concrete interface of a web service. A service can specify several concrete network addresses for accessing functions of the service, such network addresses are defined using `endpoint` elements. In addition, `endpoint` elements reference `binding` elements which contain detailed information about the transmission of data, e.g. the protocol to be used for the current endpoint.

- The abstract level: The `operation` element represents an operation offered by the web service and defines the messages that need to be transmitted between server and client in a specified sequence in order to access that operation. Each (concrete) `binding` element references all (abstract) `operation` elements it supports. `Operation` elements are grouped together inside an `interface` element.

For analyzing WSDL’s applicability as user interaction description language in AAL environments, the abstract part is most important, since it defines interactions between client and

server: A similar pattern could be used for describing the communication between the user through user interface devices and AAL services.

The core component of WSDL seen as interaction description language is the `operation` element. Each `operation` element defines a sequence of messages that are transmitted between client and server, and a `pattern` attribute specifying the message exchange pattern the operation uses. Part 2 (”Adjuncts”) of the WSDL specification contains several pre-defined message exchange patterns that may be used as content for this attribute. Examples include the `in-only` pattern that consists of exactly one message that is sent from the client to the server, and the `in-out` pattern that consists of exactly two messages, one sent from the client to the server, and one from the server to the client as feedback to the first message. However, custom message exchange patterns may be defined and used alternatively.

A message itself is defined inside an `operation` element by using either the `input` or the `output` element. Both elements define a single message and its content. The structure of a message may be defined in an extra section of the WSDL document (inside the `types` element) using XML Schema [43]. The naming follows the point of view of the server, so `input` defines a message sent from the client to the server, and `output` a message sent from the server to the client. The sequence of input and output message inside an `operation` element must match the message exchange pattern referenced by the operation.

In addition to those in and out messages, fault messages may be defined by the `infault` and `outfault` messages. Their purpose is to inform one partner in the communication (either client or server) that an error in the service execution has occurred. The relation between standard messages and fault messages (e.g., if a fault message replaces a standard message on occurrence) is defined in a special ruleset, defined as part of the message exchange pattern the current operation uses.

Information flow and rendering

In contrast to a user interface description language that defines interface elements to be manipulated by the user, WSDL is an interaction description language that describes the communication between two endpoints: client and web service (although this description pattern could also be applied to communication between the user and any service through the use of a user interface device). In practice, a client can read the WSDL document provided by a web service, in order to determine which operations the web service offers and how they can be accessed. To actually call the operations, SOAP or any other protocol is used. Therefore the WSDL specification contains no information about the generation and rendering of user interfaces. In almost all practical use-cases, a static Browser-based user interface (implemented using HTML, Java applets, etc.) is used for accessing the web services specified by WSDL.

Analysis

The following enumeration contains a detailed analysis of the Web Services Description Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   By defining interactions rather than user interfaces which can be accessed trough any user interface on any device, WSDL provides a very high level of abstraction. The use of WSDL descriptions, however, is not sufficient to allow the automatic generation of multi-modal user interfaces. In addition to WSDL, a user interface description language is needed that implements the interactions defined in WSDL and maps them to user interface commands.

2. **Adaptability**

   Since WSDL does neither define nor generate user interfaces, no information about the adaptability can be provided.

3. **Openness**

   The full specification of all WSDL language versions (including the current 2.0 version) is available for free, published by the W3C.

4. **Organizational background**

   The WSDL specification is published and worked on by the World Wide Web Consortium (W3C), an international community developing open Internet standards, thus ensuring wide acceptance and implementation of the WSDL standard.

5. **Status**

   The latest version (WSDL 2.0) was published as recommendation by the World Wide Web Consortium (W3C) in 2007 [17]. The 2.0 version is the successor of the widely used WSDL 1.1 specification and currently used in many SOAP-based web service applications.

6. **Number of implementations**

   Due to the close interconnection with the popular SOAP protocol, it can be assumed that WSDL is used in many web service applications on the web, although the exact number cannot be identified. As mentioned before, to allow auto-generation of user interfaces based on WSDL interaction descriptions, additional UI generators are necessary. One example of such an implementation is part of the Java API for XML Web Services (JAX-WS) Reference Implementation[7]. The *wsimport* toolbox generates Java source code out of a given WSDL document. It does not automatically generate full user interfaces, but the main Java code artifacts that provide a basis for developing a full web application, thus saving development effort and time.

7. **Number of supported target platforms**

   As mentioned above, WSDL only describes interactions, not user interfaces. Therefore it theoretically supports all modalities and all platforms, constrained only by the concrete user interface generator used as supplemental component.

[7] `http://jax-ws.java.net/`

level of abstraction:	very high
adaptability:	
accessibility:	N/A
context-awareness:	N/A
use-case awareness:	N/A
openness:	high
organizational background:	open standards community
status:	W3C recommendation
implementations:	many
supported target platforms:	all

Table 2.5: Comparison chart for the Web Services Description Language (WSDL)

2.2.6 Web Service Experience Language (WSXL)

The Web Service eXperience Language (WSXL) was developed to reduce development effort while building web applications for different distribution channels by re-use. Service-based web applications specified using WSXL can easily be adapted to different presentation scenarios. This means that one web application can be accessed in different ways without the need of redevelopment, for example directly on a web page, through a web portal, or embedded in a third party's web client. In addition, this approach allows the re-use of small web services in several web applications. WSXL is based on or closely collaborating with existing, widely-used web standards, such as the SOAP protocol, the Web Services Description Language (WSDL), and the XForms standard [13].

Language specification

The WSXL specification [10] resembles the widely-used Model-View-Controller pattern, as it separates presentation, data, and control in particular components. A WSXL application consists *"of one or more data and presentation components, together with a controller component which binds them together and specifies their interrelated behavior"* [16].

WSXL components

A *presentation component* represents a complete user interface (e.g., a HTML page) and all of the user interface elements present on that page. Unlike other user interaction description languages, WSXL does not restrict the set of allowed UI elements. Moreover, it does not even provide a set of default UI elements, but encourages the use of elements defined in any existing widget toolkit. This ensures maximum flexibility in the generation of concrete user interfaces: For each target platform that shall be supported by a web application, one presentation component has to be defined, using UI widgets that are supported by the particular platform.

A *data component* contains all data stored by the web application that shall be available

to the user interface layer. Data components may be bound to presentation components by using control components to specify which data item is bound to which presentation element. In addition, data components may be bound to external data sources, although the technical implementation of such data bindings is not specified by the WSXL standard.

A *control component* manages the binding between presentation and data components. Its main function is to invoke event handlers in order to synchronise data that has been changed between the presentation layer and the data layer. It is also responsible for all necessary conversions of data types between the two layers.

Technically, all types of WSXL components (presentation, data and control components) are derived from the basic *base component*, therefore they all share the basic operations specified in the base component. Among these are life cycle operations that allow to explicitly create or destroy component instances, as well as export operations that transform the component to several output markup languages. In addition, each WSXL component may refer to an adaptation description that encodes detailed information about the rendering of components on different output channels. The next section describes the concept of adaptation descriptions in detail.

Adaptation

Adaptation is an important concept in the WSDL specification, and one of the most powerful concepts allowing the seamless integration into different delivery channels. WSXL includes a full Adaptation Description Language, offering service providers detailed control over several aspects of the automatically generated final user interface.

WSXL's Adaptation Description Language supports the definition of *adaptation points*. Such an adaptation point refers to an element in either a data or a presentation component. XPath [19] or XQuery [11] can be used for this reference, for example. In addition, an adaptation point specifies an operation, this might be either ”lookup”, ”insert”, or ”replace”. It means that the specified operation is permitted on the respective UI element or data item. When referencing a web service by embedding a WSXL component in any third party client software, the author of the client software is allowed to adapt the given component in the ways that are specified by adaptation points.

One example for usage of this adaptation concept is a WSXL component embedded in a third party web page: The author of the web page may exchange the CSS style attribute of one item of the presentation component for seamless integration into the web page design, if a *replace* adaptation point was specified for this item by the service provider. Similarly, a third party distributor of a WSXL component may add additional constraints to a data item of the data component, if the *insert* operation on this data item is allowed by the service provider.

Analysis

The following enumeration contains a detailed analysis of the Web Service Experience Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   Although WSXL is designed for web services and their typical delivery channels (HTML pages and Flash applets in web sites, portals, etc.), the level of abstraction is quite high. By defining separate presentation components for each output toolkit and just binding them to the underlying data components, theoretically every toolkit and therefore every output modality is supported. The approach, however, does not really generate final user interfaces automatically, a concrete user interface must rather be created manually for each toolkit the web service shall support.

2. **Adaptability**

   WSXL provides a powerful adaptability description framework, allowing in-depth sepcification for both service providers as well as distributors. Parts of user interfaces may be explicitly authorized for adaptation, and those parts may be adapted directly on code-level while all the others stay closed. This concept was designed for adaptation to different distribution channels inside the usage context of web services. Therefore no built-in support for the automatic adaptation following environmental contexts, use cases, or user preferences is provided, although it could easily be added by external toolkits.

3. **Openness**

   The full WSXL specification of both the current language version and the former, deprecated version, is freely available from the developer's web page[8].

4. **Organizational background**

   WSXL was developed by IBM.

5. **Status**

   The first language version was published in October 2001, version 2 was published in April 2002. Version 2 is the latest version, no updates have been published since then by IBM. The current development status is unknown.

6. **Number of implementations**

   Although the exact language specification is freely available, there exists no public information about the use of WSXL in practice. The language specification provides examples of WSXL implementations for two target toolkits: HTML and Adobe (former Macromedia) Flash.

7. **Number of supported target platforms**

   There exists no public information about the use of WSXL on different platforms in practice.

[8] `http://www.ibm.com/developerworks/library/specification/ws-wsxl/`

level of abstraction:	high
adaptability:	
accessibility:	no
context-awareness:	no
use-case awareness:	no
openness:	high
organizational background:	industry
status:	latest version from 2002
implementations:	at least2
supported target platforms:	unknown

Table 2.6: Comparison chart for the Web Service Experience Language (WSXL)

2.2.7 User Interface Extensible Markup Language (UsiXML)

The USer Interface eXtensible Markup Language (UsiXML) is an XML-based markup language that can be used to define user interfaces in an abstract way, independent of modalities. Its main goal is to reduce the effort necessary for generating user interfaces to be used in multiple contexts of use. It addresses mainly designers, in contrast to traditional UI toolkits that are often integrated into programming frameworks and therefore aim at developers.

Characteristics of UsiXML

The central concept of UsiXML is that it is based on a transformational approach: A user interface is described in UsiXML by combining multiple user interface models. Those models are together transformed in several steps, beginning from abstract task descriptions to concrete final user interfaces. Those steps resemble the development process of user interfaces in software development projects.

This model-based transformational approach is achieved by following these principles [26]:

- A suite of models is used to describe a specific user interface, those models being readable and editable (and therefore analyzable and manipulable) in an automatic way.
- All UI models are expressed using a common user interface description language, and they are all stored in a central model repository, ensuring compatibility and re-use.
- Development paths used in a software development project are resembled by one or several transformational steps. They can be interchanged or executed in different order to achieve different tasks in UI development, allowing flexible development approaches.

Language specification

UsiXML defines user interactions on four layers resembling transformational steps from abstract to concrete user interfaces, based on the Cameleon reference framework [15]: Tasks

and concepts, abstract UI, concrete UI, final UI. The following section explains these layers in detail [25].

1. **Tasks and concepts:** Abstract task descriptions describe the tasks a system offers the user to carry out. They define all interactions that might be carried out by the user as well as the objects manipulated in the course of these interactions. On this layer, three types of models are defined:

   - *Task models* contain tasks and task relationships. Tasks are described by their action type, frequency and importance. Action types are necessary to qualify the general nature of a task, while frequency and importance values are measured on a numeric scale and can be used later on to adapt user interfaces to different contexts of use.
   - *Domain models* describe the real-world concepts tasks resemble from the user's point of view. UsiXML uses UML diagrams to express domain models. Classes, attributes and methods are used as elements of these diagrams, as well as relationships between the various elements. They resemble objects that are manipulated by the user in any way in the course of an interaction.
   - *Context models* describe environmental settings that influence the execution of a task. This includes *user models* that assign preferences to a group of users, *platform models* that comprise capabilities of user interface devices, and *environment models* that describe all external influences (e.g. physical or psychological conditions).

2. **Abstract user interfaces** define user interactions in a modality-independent way, based on an underlying task description. An abstract UI consists of *abstract interaction objects* (AIO) that represent modality-independent forms of traditional UI widgets. Each AIO is assigned one or more functionalities, including input, output, navigation, and control functionalities. In addition, relationships between AIOs are defined. For example, similar AIOs may be grouped together, and those groups may be structured hierarchically.

3. **Concrete user interfaces** concretise abstract UIs to work with one specific input/output modality. In summary, they define the look and feel of a certain user interface, in a modality-dependent but toolkit-independent way. Concrete UIs are composed of *concrete interaction objects* (CIO) that represent UI widgets. For example, when using graphical user interfaces, widgets such as buttons, text boxes, etc. are used as CIOs. Spatial or temporal relationships between widgets may be defined, although not in absolute numbers. Graphical UI layouts, for example, are specified by using hierarchical groups of CIOs instead of using absolute coordinates.

4. **Final user interfaces** are concrete UIs that are supposed to be run on a specific platform, meaning they are platform- and toolkit-dependent. Concrete UIs must be transformed to final UIs to be rendered. Final UIs may either be interpreted at runtime (e.g. when using HTML) or compiled (e.g. when using Java Swing toolkit).

To formalize the transformation steps necessary for conversion between two of the above-mentioned abstraction layers, UsiXML proposes a graph-based transformation specification.

The advantage of this approach is the presence of a mathematical formalism that allows easy verification of transformation steps. An additional benefit is that graph-based transformations can easily be visualized [26].

Such a graph-based transformation logic uses a tree-based structure: The abstract task description (the first abstraction layer) is used as root of the tree, the corresponding abstract UI forms the second level, and on the concrete UI and final UI layers different branches represent the various UIs for different modalities or platforms, respectively.

All of the four abstraction layers can be further decomposed into sublayers. This means that transformations between two neighboring layers include several transformation steps. This way transformations between any of the four layers can be realized by composing multiple transformation steps and carrying them out on after the other.

In general, transformations can be applied in both directions, either from an abstract description to a more concrete one, or from a concrete to an abstract description. For example, an existing graphical UI can be converted to a voice-based UI using UsiXML: First, the existing final UI is stepwise abstracted by walking up the transformation tree towards the root until the abstract UI layer is reached, and then the abstract UI is gradually transformed to a concrete voice-based UI by traversing the tree in the opposite direction.

Carrying out transformations is impossible without detailed mapping information, encoding which elements of the source layer are to be transformed to which elements of the target layer. This information is included in special rule sets. For example, for each UI toolkit that should be supported (meaning that a concrete UI can be transformed to a final UI for this toolkit), such a rule set must be provided from an external source.

Analysis

The following enumeration contains a detailed analysis of the User Interface Extensible Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   Due to the layered structure of UsiXML, the language provides a high level of abstraction. In theory, all modalities are supported since the first two abstraction layers, tasks and abstract user interfaces, are modality-independent. In practice, there are two restrictions concerning device-independence: First, UsiXML is currently designed to support only two types of modalities, as stated in [25]: *"Two modalities lie in the intended scope of USIXML: graphical and auditory."* However, due to its structure the language should be quite easy to extend to support additional modalities. Second, only target toolkits are supported that provide a rule set for transformation of concrete UIs to final UIs in that toolkit.

2. **Adaptability**

   Full adaptability of user interfaces defined in UsiXML is supported. Context models are use to store information about user preferences, device capabilities, and environmental

settings, thus offering a complete framework to support automatic accessibility, adaptability to contexts of use, and use-case awareness.

3. **Openness**

   The full language specification, including documentation, examples, and tools, is freely available from the UsiXML project web page[9].

4. **Organizational background**

   UsiXML was developed by a research group with members from several universities and research organizations.

5. **Status**

   Work on UsiXML began in 2004. The latest version 1.8 was published in 2007, current status of development is unknown.

6. **Number of implementations**

   Several graphical editors for creating UsiXML documents were published, including plugins for generating final user interfaces in various target toolkits. One of those editors is GraphiXML [29], which supports XHTML, Mozilla XUL (see section 2.2.12), and Java as target platforms. In addition, several rendering engines for UsiXML documents are available, supporting Adobe Flash and OpenLaszlo, among others.

7. **Number of supported target platforms**

   As mentioned before, the current version of UsiXML is intended for use with only two modalities: Graphical and voice-based user interfaces.

level of abstraction:	rather high
adaptability:	
accessibility:	yes
context-awareness:	yes
use-case awareness:	yes
openness:	high
organizational background:	research organization
status:	latest version from 2007
implementations:	many
supported target platforms:	2

Table 2.7: Comparison chart for the User Interface Extensible Markup Language (UsiXML)

[9] `http://www.usixml.org`

2.2.8 User Interface Markup Language (UIML)

The User Interface Markup Language is a very powerful framework that has been around for nearly 15 years and is still being developed further. UIML is a meta-language, not specifying concrete UI elements on its own, but providing a framework for the definition of custom vocabularies that can then be used to create generic user interface descriptions.

Development on UIML was started in the late 1990s, facing one major problem in traditional UI development: User interfaces were defined using imperative programming languages, such as C, C++, Java, etc., which were originally designed and therefore more suited for describing application logic, not user interaction concepts. At the same time, the HTML 4.0 standard [41] was published as the first HTML version that separated hypertext markup from presentation styles. The development of UIML was inspired by the generic concept of HTML, which provided one markup language that could be rendered on several platforms [2].

Based on this, UIML's main goal is helping UI developers in creating user interfaces that are sufficiently generic to be used on different platforms, thus significantly reducing the effort in developing multi-platform user interfaces. User interfaces defined using UIML are either automatically transformed to different target languages, or interpreted and rendered on target devices in real-time.

Structure of UIML

Defining a user interface in UIML resembles answering six major questions regarding the design of the user interface and its connection to the underlying application logic [22]:

1. In which parts can the UI be deconstructed in order to represent the structure of the UI?
2. What is the content associated with each of the parts?
3. Which style is used for presentation of each of the parts?
4. What is the behavior of each of the parts when the user interacts with the UI?
5. Which APIs can be used to connect to external components, such as application logic and data sources?
6. How is the UI mapped to target toolkits?

The six parts of a user interface description resembled by those six questions build the basic structure of UIML. It is based on the *Meta-Interface Model* (MIM) suggested by [37], which is defined in a hierarchical manner: On the first level, MIM separates the user interface from underlying application logic and data sources, and from the presentation on specific devices:

- The *logic* component encapsulate the business logic the underlying application provides. It offers the user interface access to the application logic while hiding implementational details such as method and variable names or the protocols used for communication.

- The *presentation* component includes information about the rendering of the final user interface, hiding details such as widgets, attributes and event handling.
- The *interface* component allows the description of the communication between user and application in a device- and application-independent way.

On the second level, the *interface* component is further divided into four sub-components, separating content, style and behavior:

- The *structure* component defines parts that comprise the UI, allowing to group UI elements together and structure them hierarchically.
- The *content* component defines the content of each part. This may include content types such as text, image, sound, etc.
- The *style* component defines the exact presentation of each part, including attributes such as font family, text size, colors, etc.
- The *behavior* component defines for each part the events, conditions and actions required when interacting with this part of the UI.

Figure 2.1 presents a schematic illustration of MIM's structure.

In summary, UIML separates a user interface into six parts: Structure, content, style, behavior, APIs to external components, and mappings to UI toolkits, each of which is related to one of the six above-mentioned UI design questions.

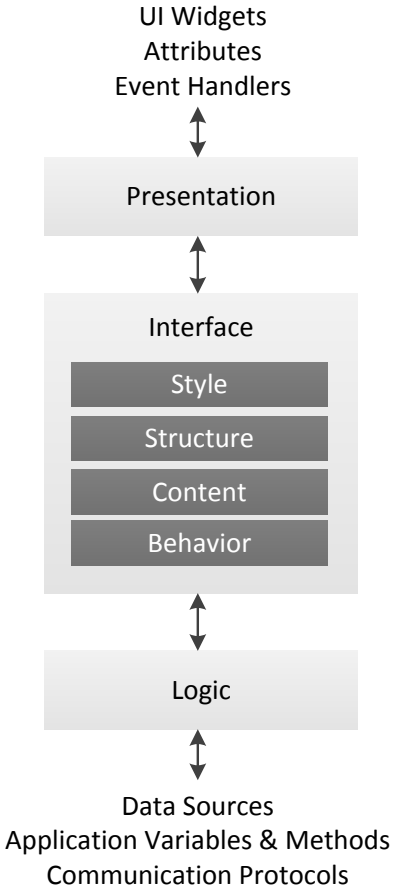
Language specification

UIML is specified [22] as an XML-based meta-language, as it does not contain a predefined set of XML tags that represent UI elements, neither concrete UI widgets nor abstract interactor elements. Instead, it relies on external vocabularies that define a set of abstract UI elements and their mapping to concrete, platform-dependent UI widgets. A UIML document references such a vocabulary, objects defined in the vocabulary may then be used in the UIML document.

Each vocabulary may have a different level of abstraction, depending on the number of target platforms and modalities it shall support. It must define each target toolkit its supports, and each element defined by the vocabulary should include a mapping to one concrete element per toolkit that is targeted. In some cases, an abstract element may map to different concrete elements, or to a certain combination of several concrete elements. In addition, some target toolkits may not support or require certain elements, in this case an abstract element would have no mapping for this particular target toolkit [9]. Vocabularies that are defined to support only one target toolkit, or a limited set of target toolkits that are similar and use the same interaction modality, may be built less generic than vocabularies that need to support a high number of modalities. However, such little generic vocabularies can not be extended when a new target shall be supported later on.

Besides a `head` element that contains metadata about the document and a `template` element that allows the definition of UIML fragments for re-using them, each UIML document

Figure 2.1: Components of the Meta-Interface Model (MIM) used by the UIML language. (Adapted from figure 4-7 in [37])



may contain two major elements: An `interface` element containing the structure of the user interface and its content, style and behavior, and a `peers` element containing references to a vocabulary and information about the connection to the application logic.

The `peers` element may contain `presentation` and `logic` elements. The `presentation` element references a vocabulary, which might either be a predefined vocabulary (several of which are available online for common use, for example vocabularies that support HTML and Java Swing as target toolkits), or a custom vocabulary. The `logic` element acts as connection to the underlying application. It contains information about the calling conventions of methods specified by the application that provides the business logic the user interactions are based on.

The `interface` element may contain four types of subelements, as described in section 2.2.8:

- The `structure` element references UI elements defined by a vocabulary, and defines their initial relations to each other at the time when the user interface is rendered. Since the

specification shall be generic, these relations may be spatial (as needed in graphical UIs), temporal (as used in speech-based UIs), or of other nature. The `structure` element contains several `part` elements which either refer to UI elements or act as containers that are used to group other elements. `part` elements may be nested to create a hierarchical structure of UI elements.

There may be more than one `structure` element present inside an `interface` element. In this case, each `structure` element contains the UI's structure for one target platform or modality, and it must specify this target in a special attribute. The decision on which `structure` element to use lies in the responsibility of the rendering mechanism on the end device and is not in the scope of the UIML specification.

- The `content` element references one `part` component defined in the `structure` section of the UIML document. In addition, it contains the content of the respective UI element. The content may be of several types such as text, image, or sound, but the set of supported content types is not predefined and not limited by the UIML specification.

  Multiple `content` elements may reference the same `part` element. This is used for defining several contents for one UI element, which might be useful for providing the content of text-based UI elements in several languages, for example. The decision on which `content` element to use lies in the responsibility of the rendering mechanism on the end device and is not in the scope of the UIML specification.

- The `style` element defines a style sheet used for rendering a UI element, similar to the way CSS style sheets are referenced by HTML elements. Each `style` element contains several `property` elements, encoding key-value pairs for one specific attribute of a UI element.

  In most cases there will be more than one `style` element present inside an *interface* element. This allows to provide separate style sheets for each target platform or modality, or even several style sheets for the same target platform. The latter can be useful since a group of similar devices that share the same platform may differ in their hardware capabilities (for example, a mobile phone that features only a monochrome display might need a different style sheet than a mobile phone offering a color display).

- The `behavior` element contains several `rule` elements, each of which defines one rule for the behavior of UI elements. This includes all actions that are invoked when the user interacts with a UI element. Rules consist of `condition` elements and `action` elements. If all conditions defined inside a `rule` element (for example, a certain button was pressed by the user), then all actions defined inside the same `rule` element are executed (for example, a method of the underlying application is called).

Information flow and rendering

Abstract user interfaces defined as UIML documents need to be rendered on end devices in order to be presented to the user. UIML provides to types of rendering:

- **Compiling**

  UIML definitions are first translated to different target languages. The selection of target languages depends on the languages supported by the target devices the user interface shall be used with, and by the vocabulary the UIML definition uses. The resulting packages are transferred to a target device and rendered there, like any native application.

- **Interpreting**

  An end device that is capable of parsing UIML and understands the vocabulary used by a UIML document may directly process and render UIML definitions. In this case, the UIML document is transferred to the device and a local interpreter installed on the device reads it and uses an API to display the user interface and capture user interactions.

Analysis

The following enumeration contains a detailed analysis of the User Interface Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   As mentioned above, the level of abstraction is completely dependent on the vocabulary used by a UIML definition. When using a vocabulary for only one target language, the user interface elements used in UIML may be quite concrete. When defining a vocabulary that shall support a variety of different target modalities, UI elements will automatically be more generic.

   Unfortunately, the level of abstraction can not automatically be retrieved from UIML definitions or vocabularies, since even vocabularies covering only one target language might be defined in a very generic way. In addition, the level of abstraction also depends on the number of interface subcomponents that are bound to specific platforms: Support for a certain platform might be very low-level even if the vocabulary supports that platform if there are no specific `structure`, `content` and `style` elements defined in the UIML document.

2. **Adaptability**

   UIML provides the possibility to define different structures, style sheets, and contents as part of one user interface description. This option can be used for reacting to user preferences, contextual settings, use cases, etc. The mechanism of choosing one of the definitions, however, is beyond the scope of the UIML specification and is left to compilers and interpreters. Therefore UIML provides no built-in functionality for user interface adaptation.

3. **Openness**

   The current UIML language specification [22] as well as the previous one can be downloaded for free from the web page of OASIS[10], the standardization organization that is

[10] `http://www.oasis-open.org/committees`

responsible for publication and further development of the UIML standard. In addition, the specification states that it may be freely implemented by anyone.

Previous versions of the specification can be downloaded for free from the former UIML web platform that is now discontinued[11].

4. **Organizational background**

   Development on UIML was started in 1997 at the Virginia Polytechnic Institute and State University. To further develop the core language and additional tools, a spin-off corporation called "Harmonia" was founded [45]. The UIML standard has been adopted by the Organization for the Advancement of Structured Information Standards (OASIS). The OASIS UIML committee is now responsible for further development concerning the UIML standard.

5. **Status**

   The first version of UIML was published in 1997, the language has been further developed since then and refinement is still in progress. The latest version is 4.0 which was published as a final specification in 2009.

6. **Number of implementations**

   Many different UIML compilers and interpreters have been developed in the last 13 years, covering a variety of target language and toolkits. For some of those, several implementations are available. Supported target languages and toolkits contain HTML, Java, C++, .NTE, QT, Symbian, WML, VoiceXML, and others.

7. **Number of supported target platforms**

   The above-mentioned implementations cover at least four different platforms: Desktop PCs, mobile devices, multimedia devices (TVs) etc., and speech-based systems. However, there may be other implementations supporting additional platforms.

2.2.9 Dialog and Interface Specification Language (DISL)

The Dialog and Interface Specification Language (DISL) [42] is an extended subset of the UIML language specification (see section 2.2.8). It provides a modeling language for specifying dialog models in an abstract way that can be used to generate user interfaces for multiple modalities and platforms.

[11] `http://uiml.org/specs/index.htm`

level of abstraction:	medium
adaptability:	
accessibility:	prepared
context-awareness:	prepared
use-case awareness:	prepared
openness:	high
organizational background:	standardization organization
status:	latest version from 2009, under development
implementations:	many
supported target platforms:	at least 4

Table 2.8: Comparison chart for the User Interface Markup Language (UIML)

Language specification

DISL follows the approach of separation of control model and dialog model. The control model contains all data that represents the application state, while the dialog model is responsible for presentation and interaction. This separation has the following advantages:

- It improves the adaptability, as automatic UI adaptation needs to deal only with the dialog model while the control model can be left unchanged.
- It enables independent rendering on different platforms and modalities.
- It eases transformations to other UIDLs.

The dialog model proposed by DISL contains three parts:

- **Dialog flow** is the central component. It controls the data flow inside the user interface components. The dialog flow is a sequence of four steps that are repeated periodically and are carried out by the two remaining components:
  1. Generate a dialog
  2. Present the dialog
  3. Capture user interaction
  4. Evaluate the interaction
- The **presentation** component is responsible for steps 1 and 2 in the cyclic process presented above. It depends on *generic interfaces* (compare *abstract interfaces* in most other UIDLs) that describe the user interface to be generated in an abstract way and are specified manually prior to the dialog flow process. They consist of generic UI widgets which are modality-independent as they describe only basic operations such as trigger, data input, data output, etc. At runtime, generic interfaces are mapped to distinct modalities by

mapping each generic widget to a concrete widget. Finally the presentation component presents the generated modality-specific user interfaces to the user using a target device.

- The **interaction** component captures user interactions within the currently presented user interface and processes them. All user input is collected from the different modalities. Based on the widgets that delivered the user input, the generic widgets on which they are based are tracked using reverse mappings. On the generic interface level, properties of generic widgets can be set or events can be triggered, according to the user input.

Step 4 in the cyclic dialog flow is executed by the behavior controller: The events triggered and properties set by the interaction component are passed on to the behavior resolver which manages the control model. The control model consists of content elements which represent the current state of the user interface. The behavior resolver reacts to interactions captured by the interaction component of the dialog model by setting content elements' properties. Immediately after such an edit operation, it asks the dialog model to refresh the UI presentation.

Since widgets referenced by the presentation component are based on content elements of the control model, a changed property followed by a refresh of the presentation results in the according UI widget to be presented in a different way, reflecting the user interaction. It should be noted that the control model only contains properties that represent the content of an element, not its style, since the presentation style of a widget is controlled directly by the presentation component.

Analysis

The following enumeration contains a detailed analysis of the Dialog and Interface Specification Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   Due to the hierarchical structure and the separation of data and presentation, the level of abstraction is quite high. Every target platform and modality is supported if adequate mapping rules are provided.

2. **Adaptability**

   Adaptability is one of the key issues of DISL. The language is designed to support switching of end devices on the fly, therefore adaptability to different use cases and environmental contexts is a built-in core component. The specification of user preferences is not provided by the language framework, although adaptability to such profiles could easily be accomplished by using the built-in adaptability mechanisms.

3. **Openness**

   Besides scientific papers, no detailed information about the DISL specification is available.

4. **Organizational background**

   The DISL language was developed by a research group formed of members of the Paderborn and Kassel Universities in Germany.

5. **Status**

   DISL was proposed in a research paper published in 2006, the current development status is unknown.

6. **Number of implementations**

   No information about systems running DISL is available.

7. **Number of supported target platforms**

   No information about systems running DISL on specific platforms is available.

level of abstraction:	rather high
adaptability:	
accessibility:	no
context-awareness:	yes
use-case awareness:	yes
openness:	low
organizational background:	university
status:	latest version from 2006
implementations:	unknown
supported target platforms:	unknown

Table 2.9: Comparison chart for the Dialog and Interface Specification Language (DISL)

2.2.10 Model-based language for interactive applications (MARIA XML)

MARIA stands for Model-based lAnguage foR Interactive Applications, an XML-based user interaction description language. It mainly focuses on the definition of user interfaces used to access web service functionalities. The language follows a semi-automatic approach for generation of user interfaces: Basic final user interfaces are generated automatically from abstract user interface descriptions, but developers are given the possibility to refine these concrete interfaces. This concept allows human intervention, but reduces manual effort.

Language specification

MARIA XML introduces two abstraction layers: User interfaces are defined on an abstract level and on a concrete level [35]. In addition, *data models* are defined that represent the underlying data structure, as well as *event models* which represent the underlying application logic.

Abstract user interfaces are independent of modality and focus on interactions carried out by the user with the web service. They consist of abstract interactors which are grouped together and interconnected using relations. There are two general types of abstract interactors: Interaction objects can be used by the user to activate events, manipulate attributes of other interactors, etc. Output objects are used only for displaying data, presenting feedback, alerting alarms, etc. to the user, but the user can not interact with the web service through those objects. Both types of objects are presented to the user as part of the final user interface. Both are bound to elements in a data model that represents the data displayed and/or manipulated by the object, but only interaction objects may be bound to events defined in an event model.

Concrete user interfaces are modality- and platform-specific but toolkit-independent. They act as *"an intermediate description between the abstract description and that supported by the available implementation languages for that platform"* [35]. Each interactor on the abstract level is mapped to an element of the concrete user interface, corresponding to typical user interface widgets (e.g. button, text box, list box). These concrete elements therefore refine the abstract interactors. The mapping of abstract interactors to concrete elements depends on the characteristics of the target platform: A certain data type might be displayed using different UI widgets, but some of those widgets are more suitable for certain modalities and platforms than others.

MARIA XML mainly aims at creating multimodal user interfaces for web service. Webs service descriptions written in WSDL (see section 2.2.5) are used as task descriptions. Abstract user interfaces are built upon these task descriptions.

Transformations between abstract and concrete user interfaces are carried out using predefined rule sets. In general, those tranformations can be carried out in both directions, from abstract to concrete user interfaces as well as vice versa. In addition, MARIA also supports the rule-based transformation of concrete user interfaces to final implementations. In any case, a rule set defines mappings, either between abstract and concrete interactor elements, or between concrete elements and toolkit-specific wigdets.

In general, the rules used for carrying out transformations are predefined by developers or user interface designers prior to performing a transformation. There are two ways of specifying transformation rules [34]:

- Rule sets for general use are defined once and can be re-used for several documents. A mapping for each element on the source abstraction layer (abstract or concrete UI layer) must be provided, referencing an element in the target abstraction layer. If the transformation defined by this rule set is performed on a certain source document, those mappings are executed, replacing source elements with target elements.

- As mentioned before, developers may refine user interfaces that were automatically generated as result of a transformation. This is achieved by editing predefined mappings at

the document instance level. These edited rules are used only once for a certain document or even a subset of a document, allowing to fine-tune the final result if the general transformation produces unsatisfactory results.

Information flow and rendering

When using MARIA XML for specifying user interfaces for web services, the usual information flow is as follows:

1. An abstract user interface is defined based on the web service's WSDL description.
2. The abstract user interface is transformed to multiple concrete user interfaces, one for each target platform to be supported.
3. Each concrete user interface is transformed to multiple implementations, one for each UI toolkit to be supported.
4. The final toolkit-specific implementations are executed on the target devices.

Alternatively, the developers of the MARIA XML language suggest a migratory approach [34]. The aim of this approach is to provide a seamless experience to the user, allowing the free selection of target devices not only prior to but also in the course of accessing a web service. The user may switch between devices while using an application, and the user interface is automatically adapted to the new devices while preserving application state.

This migratory user interface solution makes use of a migration server that stores concrete UI models and controls user interface deployment. When switching to a new device, the server receives a script containing detailed information about this devices. The current concrete UI model is then semantically redesigned to present the same information in a way that is adequate to the new device. This redesign is achieved dynamically at runtime, the created user interface is then delivered to the target device and presented to the user.

Analysis

The following enumeration contains a detailed analysis of the Model-based language for interactive applications according to the criteria defined in section 2.1.

1. **Level of abstraction**

   Due to the structural distinction of task descriptions (through WSDL), abstract and concrete user interfaces and final implementations, the level of abstraction is quite high. In theory, every modality and every target toolkit is supported, if adequate rule sets have been defined.

2. **Adaptability**

   Two potential mechanisms for UI adaptation can be identified:

MARIA XML provides the possibility to influence the UI generation by overriding mapping rules in certain cases. This can be used to adapt the generated user interface according to different external factors, MARIA XML however does not offer automatic adaptation mechanisms. Also no language constructs are provided to store user preferences or device profiles.

Alternatively, the mechanism of migratory user interfaces already implements an automatic adaptation based on contexts of use, similar approaches could be used to achieve adaptation based on user preferences and use cases.

3. **Openness**

   MARIA XML was developed as part of the MARIAE (MARIA Environment) tool which can be downloaded for free after registration from the project web page[12], including source code. The full language specification however is not available online.

4. **Organizational background**

   The MARIA XML language and the MARIAE tool were developed by HIIS Laboratory, a research group focusing on human-computer interaction of the Italian National Research Council (CNR).

5. **Status**

   No detailed information is available about the MARIA XML language itself, however the first version of the MARIAE tool was published in 2010. The latest version 1.3.1 was published in August 2011. In contrast to most other UIDLs, it is a relatively young development, although MARIA XML is based on the discontinued TERESA XML language which was developed by the same research group.

6. **Number of implementations**

   The only publicly available use of MARIA XML is the MARIAE (MARIA Environment) tool[12] which helps in developing multimodal user interfaces and applications based on web services. It is unclear how many working rule sets for transforming concrete user interfaces to toolkit-specific implementations exist.

7. **Number of supported target platforms**

   MARIAE includes example scenarios that demonstrate the generation of concrete user interfaces for at least three different platforms: desktop PCs, mobile devices, and voice-based systems.

[12] `http://giove.isti.cnr.it/tools/MARIAE/`

level of abstraction:	high
adaptability: accessibility: context-awareness: use-case awareness:	 prepared through migrational UI approach prepared
openness:	medium
organizational background:	research organization
status:	latest version from 2011
implementations:	unknown
supported target platforms:	at least 3

Table 2.10: Comparison chart for the Model-based language for interactive applications (MARIA XML)

2.2.11 Extensible Application Markup Language (XAML)

The eXtensible Application Markup Language (XAML) is a declarative markup language based on XML developed by Microsoft. While XAML was created as general markup language for the initialization of structured objects, its main use is the specification of user interfaces as part of the Windows Presentation Foundation (WPF). This makes XAML the main user interface description language used by .NET applications, since WPF is part of the Microsoft .NET framework. Another area of use for XAML is the definition of user interfaces for web applications that build upon the Microsoft Silverlight framework.

Language specification

[30] gives an overview of the specification of XAML for its main use case: as user interface description language for the WPF framework. The language follows an XML-based structure with custom elements and attributes. XAML is closely coupled with the language framework that uses XAML as UIDL (in most cases .NET).

In general, all classes that are available in the underlying programming language can be used as elements, this includes not only classes that represent user interface components but even custom classes defined by developers inside the application. Properties of those classes can be referenced as attributes. Each XML element defined in a XAML document declares an instance of the class referenced by the element. Defining attributes sets the initial values of the respective properties of this object. This concept is mainly used to declare user interface components and their hierarchy, however it can also be applied to non-visual elements that are references by UI components, for example.

XAML contains several concepts to combine user interface markup with the underlying application code. First, every XAML element can be assigned a name by specifying the `name` attribute. The object can then be referenced from code through this name as if it was a local variable. In addition, event handlers can be defined by simpy referencing method names that are

defined in code, when the event occurs this certain method is executed.

Another powerful concept of XAML is data binding. It can also be used to connect UI markup to underlying code. XAML provides a special syntax to bind the attribute of an element to a resource instead of setting it directly in XAML. Such a resource can be, among others, a simple data model defined in underlying code, i.e. a class that contains member variables holding values that shall be used as attributes for XAML elements. The advantage compared to attributes set directly in XAML is that changes on one end of the data binding are automatically propagated to the other. This means that whenever the content of one variable in the data model changes, the user interface is automatically adapted since also the attribute of one UI component has changed. This works also vice versa, e.g. when the user changes the content of a UI widget the underlying data model can be updated automatically. The direction of automatic propagation of data can be defined (either from the UI to the data model, or from the data model to the UI, or in both directions).

Information flow and rendering

XAML is usually compiled to binary code, integrated into .NET applications and executed on end devices through the .NET framework. This ensures that no compatibility or rendering problems can occur. At the same time however it reduces the amount of supported end devices to those that run the .NET framework.

The only exception of this rule is the use of XAML in Silverlight web applications. In this case, XAML code can be sent to the browser and is interpreted at client side by the Silverlight browser plugin. The advantage of this approach is that search engines can scan the whole source code of a web application, which is not the case for compiled code [31]. This is also one characteristic that distinguishes Silverlight from competitive solutions, first of all Adobe Flash, since web applications created using Flash or MXML (see section 2.2.13) are always delivered as binary applications.

Analysis

The following enumeration contains a detailed analysis of the Extensible Application Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   XAML relies on the user interface components specified by the underlying language framework. Since it is mainly used in combination with either WPF or Silverlight and those framework define rather concrete UI widgets (such as *button*, *textbox* etc.) that are modality- and toolkit-dependent, the level of abstraction is quite low. Also the implementational details indicate a low abstraction level, since user interfaces specified using XAML can only be rendered on devices supporting the .NET toolkit.

2. **Adaptability**

   Nearly all attributes of XAML user interface components can be bound to variables and

therefore changed at runtime instead of setting them explicitly before compiling. This concept provides the basis for adaptation of existing user interfaces, XAML however does not provide mechanisms to store users' preferences or to react to environmental factors. The issue of different user interfaces on different devices is not crucial to XAML, since in most cases a separate user interface for each type of device needs to be created manually anyway.

3. **Openness**

   XAML was published by Microsoft as an open standard, the full specification is available online.

4. **Organizational background**

   XAML is developed by Microsoft.

5. **Status**

   A separate XAML version is released with each new version of the .NET framework as well as with each Silverlight version. The latest versions are .NET 4.5 (published only as developer preview in 2011) and Silverlight 5 (published as release candidate in 2011). It can be assumed that XAML will be further developed with newer versions of .NET.

6. **Number of implementations**

   Support for XAML is built into WPF and Silverlight, as mentioned above. Beside those two platforms, there exists the Mono project that aims at developing an open source, cross platform implementation of the .NET framework. The Olive project[13] provides add-on libraries to the Mono framework, including libraries that add XAML support. The Moonlight project[14] aims at developing an open source implementation of Silverlight as part of Mono.

7. **Number of supported target platforms**

   Both .NET and Mono are available for desktop PC as well as mobile platforms. In addition, XAML supports web-based platforms through Silverlight.

2.2.12 XML User Interface Language (XUL)

The XML User interface Language (XUL) is a user interface description language developed by Mozilla. It is based on XML and not compiled but interpreted at runtime by a special rendering engine. It's main use is the definition of user interfaces for applications developed by the Mozilla community, although also some additional projects make use of it.

[13] `http://www.mono-project.com/Olive`
[14] `http://www.mono-project.com/Moonlight`

level of abstraction:	low
adaptability:	
accessibility:	no
context-awareness:	no
use-case awareness:	N/A
openness:	high
organizational background:	industry
status:	under development
implementations:	4
supported target platforms:	3

Table 2.11: Comparison chart for the Extensible Application Markup Language (XAML)

Language specification

According to [20], XUL separates the description of user interface objects and of these objects' styles. Similar to HTML, in XUL user interfaces are defined in a hierarchical tree-like structure, as some objects may contain others. Since the user interface description is not compiled but interpreted, the rendering engine builds a document object model (DOM) and hierarchically renders its elements.

XUL provides a predefined set of user interface widgets. XUL supports only graphical user interfaces, which allows the set of UI widgets and their available attributes to be relatively small and concrete. There are predefined widgets for typical use cases of graphical user interfaces, such as button and checkbox. In addition, there are also advanced widgets such as datepicker or colorpicker that allow to solve relatively complex tasks easily.

In addition, HTML elements can even be used as part of XUL documents just as native XUL user interface widgets. This can be useful for referencing elements XUL does not provide, such as tables for creating advanced layouts, and for embedding Java applets and similar external content.

User interface elements defined in a XUL document are positioned automatically in order to fill the available screen space. This way user interfaces automatically adapt to different screen resolutions, e.g. on different hardware devices. Elements' width and height can be set explicitly, but not their exact position. Elements however can be grouped by using `box` elements. All elements defined inside a `box` container are displayed next to each other either horizontally or vertically. The orientation (horizontal or vertical) can be set through a special attribute.

Similarly, `stack` and `deck` containers can be used as simple layout managers, these position all child elements on top of each other instead of next to each other. In this case, each elements exakt position inside the container can be defined.

As mentioned above, XUL supports separation of user interface description and styling. Styles are defined using Cascading Style Sheets (CSS) [12]. Similar to HTML pages, CSS documents can be referenced by XUL user interface descriptions. Inside a style sheet document, each XUL element's attributes can be set.

The application logic is defined in JavaScript documents that are referenced in the XUL

document. Each XUL user interface widget provides several event listeners that can be assigned methods defined in the referenced JavaScript. Alternatively, such method invoking commands can be put in special `command` objects, and these commands can then be referenced inside an event handler instead of calling JavaScript methods directly. This has the advantage that all references to external JavaScript code can be collected in a separate section of the XUL document (e.g., if all `command` elements are declared at the bottom of the document). In addition, re-use is possible as several UI elements can call the same command.

Analysis

The following enumeration contains a detailed analysis of the XML User Interface Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   XUL focuses on graphical user interfaces and provides widgets that can only be used in graphical environments, therefore the level of abstraction must be defined as low.

2. **Adaptability**

   XUL does not integrate mechanisms for defining user preferences or contexts of use, therefore XUL-based user interfaces can not automatically adapt to those settings, although adaptation could be integrated using the flexible CSS styling system that allows redesign of all user interface elements at runtime.

   XUL user interfaces automatically adapt to different hardware capabilities such as screen resolutions. However this applies only to simple dynamic resizing and positioning of widgets, since XUL is restricted only to graphical user interfaces. In addition, XUL does not support the automatic display of different user interfaces (accomplishing different tasks) on different devices, since there is no way to define the type of tasks each widget implements.

3. **Openness**

   The full specification is available at the Mozilla project web page.

4. **Organizational background**

   XAML is developed by the Mozilla open source community.

5. **Status**

   XUL is used as user interface description languages by all applications published by Mozilla, the most common ones being the *Firefox* web browser and the *Thunderbird* e-mail client. There are some third-party applications however which also make use of the XUL language, among them the movie/theater project management tool *CeltX* and the instant messaging client *Instantbird*.

   In addition, the XUL language is used to define user interfaces of Firefox and Thunderbird add-ons. Finally, XUL can also be used to develop applets to be embedded into web

pages [28]. This approach however is not very common among web developers, mainly due to the restricted browser support (XUL applets can only be run in Mozilla-based browsers that make use of the Gecko rendering engine).

6. **Number of implementations**

   Currently there is only one XUL interpreter available, namely the Gecko rendering engine developed by Mozilla. This rendering engine is also used in Mozilla-based browsers to render HTML content. Standalone XUL application are interpreted and rendered by the XULRunner engine, which internally uses the Gecko rendering engine. browsers that use the Gecko rendering engine).

7. **Number of supported target platforms**

   XUL was developed solely to support graphical user interfaces.

level of abstraction:	low
adaptability:	
accessibility:	no
context-awareness:	partly
use-case awareness:	no
openness:	high
organizational background:	open source community
status:	under development
implementations:	1
supported target platforms:	1

Table 2.12: Comparison chart for the XML User Interface Language (XUL)

2.2.13 Macromedia Extensible Markup Language (MXML)

The Macromedia eXtensible Markup Language (XAML) is a declarative markup language based on XML originally developed by Macromedia, now used by Adobe as part of the Flash product series. The main use of MXML is the specification of user interfaces of Adobe Flex applications. Flex applications are developed using Adobe's Actionscript Language and compiled to executable files of type SWF (Shockwave/Flash). Such applications can be run as web applets embedded in any web page, or as standalone applications. Web applets are run using Flash Player while standalone applications are executed by the Adobe AIR runtime environment.

Language specification

The MXML tutorial published at Adobe's developer web page [7] gives an overview of the specification of user interfaces using MXML and its interconnection with Actionscript.

Flex applications are written using the Actionscript language, defined in files of type `.as`. Adobe provides several libraries which contain useful classes that can be used by all Flex applications. User interface components are mainly defined in the `MX` and `spark` packages, `MX` components being older and gradually replaced by `Spark` components from Flex version 4 onwards. In general, all those user interface components can be referenced from Actionscript code, but in most cases it is more comfortable to use MXML.

MXML code is specified in separate files of type `.mxml` using an XML-based structure. An MXML file specifies the initial state of an application or any user interface container (e.g., windows, panels, etc.) before runtime. This mainly includes user interface components (buttons, lists, images, labels, etc.). In addition, also non-visual elements may be declared in a certain section of the MXML file, causing an object of the specified type being instantiated at runtime that can then be referenced in Actionscript code. Typical examples of such non-visual elements declared in MXML are access objects to external data sources. Also animations and transitions of a certain element's attribute are often specified using MXML, because the XML-based syntax is more comfortable than the specification in Actionscript, and because those animations and transitions can then directly be referenced by UI elements defined in the same MXML file, without the need of writing a single line of Actionscript code.

MXML introduces several concepts to combine user interface markup with the underlying Actionscript code. Every element specified in an MXML file can be assigned a unique name by specifying the `id` attribute. The object can then be referenced from Actionscript code through this name as if it was a local variable. In addition, each MX or spark element provides event handlers that can be referenced as attributes, referencing method names that are defined in Actionscript code.Whenever the specified event occurs this certain method is executed.

MXML file may also contain Actionscript code, based on two different concepts. First, MXML files may contain `<fx:Script>` tags which may contain arbitrary Actionscript code that is interpreted as any Actionscript code specified in external `.as` files. Due to reasons of clearness, the use of this approach should be limited to the definition of short event handler methods, while longer Actionscript code fragments should be defined in separate files.

The second option for the direct use of Actionscript code in MXML is the data binding concept. Instead of directly specifying values for MXML elements' attributes, Actionscript code can be provided by enclosing the code section with { and } brackets. This code may reference Actionscript variables, other elements' attributes, or fixed values, in all combinations. This is a powerful and comfortable approach for binding certain attributes (e.g., the content of a text box) to a variable defined in the underlying Actionscript application. In this case, the user interface will automatically be updated when the variable's value is changed.

For separating the declaration of UI elements from the specification of their exact style, MXML supports the use of Cascading Style Sheets (CSS) [12] for defining UI elements' appearance. Elements declared in an MXML file can be referenced in a separate `.css` file and their attributes can be set using the CSS syntax.

Analysis

The following enumeration contains a detailed analysis of the Macromedia Extensible Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   MXML relies on the user interface components provided by the `MX` and `spark` libraries. Since it was developed for web applets on the one hand and desktop and mobile applications on the other hand, these libraries define rather concrete UI widgets useful only for graphical user interfaces (e.g. *button*, *list box* etc.). Therefore the level of abstraction must be defined as low. Also the implementational details indicate a low abstraction level, since user interfaces specified using MXML can only be rendered on devices that run the Adobe Flash player or the AIR runtime environment.

2. **Adaptability**

   Nearly all attributes of MXML user interface components can be bound to variables and other components' attributes and therefore can be changed at runtime instead of setting them explicitly before compiling. This concept provides the basis for adaptation of existing user interfaces, MXML however does not provide mechanisms to store users' preferences or to react to environmental factors. Through the use of layout managers, MXML user interfaces automatically adapt to different screen resolutions by re-positioning UI elements. The display of different user interfaces on different devices however is not supported by MXML.

3. **Openness**

   MXML is a proprietary standard established by Macromedia / Adobe and only used in Adobe's products. Various tutorials explaining the use of MXML and Flexin practice are available online, both provided by Adobe and by third parties. In addition, there exist projects that aim at providing a full XML Schema representing MXML since there is no XML Schema provided by Adobe, for example the *xsd4mxml*[15] project.

4. **Organizational background**

   MXML was established by Macromedia and is currently developed by Adobe.

5. **Status**

   The latest version of Adobe Flex is 4.5 which was published in 2011. Major version 4 was published for the first time in 2010 and brought about important changes, for example the introduction of the `spark` user interface components.

6. **Number of implementations**

   Flex applications must be compiled to SWF files to be executed. For execution, only Adobe AIR runtime environment and Adobe Flash player are available.

[15] `http://code.google.com/p/xsd4mxml/`

7. **Number of supported target platforms**

   The Adobe AIR runtime environment is available for Windows and Macintosh desktop operating systems, as well as for some mobile operating systems. In addition, browser-based Flex applications can be run using Adobe Flash player.

level of abstraction:	low
adaptability:	
accessibility:	no
context-awareness:	no
use-case awareness:	no
openness:	medium
organizational background:	industry
status:	under development
implementations:	2
supported target platforms:	3

Table 2.13: Comparison chart for the Macromedia Extensible Markup Language (MXML)

2.2.14 VoiceXML

VoiceXML is an XML-based markup language used to specify user interaction with speech-based systems. The language is standardized by the World Wide Web Consortium (W3C). VoiceXML was designed with a similar goal as HTML: HTML is used to specify visual web pages, while VoiceXML allows the specification of audible web content, featuring both one-way presentations and interactions. VoiceXML is used in various commercial automatic telephone systems today.

Language specification

VoiceXML documents allow the specification of speech-based interactions between a system and its user. Those interactions contain data output from the system to the user, and data input requested by the system from the user. According to [40], VoiceXML interactions may be composed by fragments of the following types:

- Output of synthetic speech prompts
- Output of predefined audio content
- Input and automatic recognition of spoken phrases
- Input and automatic recognition of DTMF key presses

In addition, VoiceXML provides mechanisms for recording speech input, specifying telephony call control detail (e.g., call transfer and hangup) and controlling the dialog flow. The latter is especially important compared to user interfaces based on other interaction modalities: Using speech-based interfaces, exactly one type of information can be output to the user or requested from the user at a time, in contrast to graphical user interfaces, for example, that may present several data items and request data from the user at the same time. Therefore VoiceXML interactions must be specified in a distinct order, and the dialog flow throughout a sequence of several interactions must be defined in detail.

A VoiceXML application consists of several VoiceXML documents, each defining one dialog. A dialog consists of several input and output interactions that are carried out. Each document specifies the dialog that shall be carried out next, by referencing the URL of the document containing that dialog.

To allow re-use of components, sub-dialogs may be defined. These can be referenced from inside any dialog, in which case the sub-dialog is carried out before returning to the original dialog. For example, a simple confirmation sub-dialog may be defined that is referenced after each user input to confirm that the data was input by the user in the correct format.

Dialogs are composed of forms and menus. A menu presents the user with several options to choose from and specifies which dialog to carry on with on each option. A form defines several interactions, each consisting of a prompt that is output to the user, the expected data requested from the user, and rules for evaluating the input data.

For specifying the format of the requested input data in detail, grammars may be defined. Usually, each dialog specifies the grammar to be used for all forms appearing during the dialog flow. In addition, a dialog can be flagged to indicate that its grammar shall be inherited to and used in subsequent dialogs. This can be useful to ensure that sub-dialogs use the same grammar as the main dialog, for example.

For additional flexibility in defining the dialog flow, variables may be declared and used throughout one dialog. One potential use case of this concept is the definition of a counter that ensures that a prompt for user input is repeated a certain number of times if the user inputs data in the wrong format.

Finally, VoiceXML offers a basic exception handling system through events. Whenever the user does not answer a prompt or inputs data that does not follow the requested format, events are thrown. Developers can specify event handlers to react to such exceptional states in order to ensure the dialog flow is not interrupted.

Analysis

The following enumeration contains a detailed analysis of VoiceXML according to the criteria defined in section 2.1.

1. **Level of abstraction**

   VoiceXML follows a quite generic approach by specifying interactions between user and system as data items that are sent between the two communication partners. VoiceXML however focuses especially on the exchange of audio content and includes functionality

only for processing speech and spoken phrases, therefore the overall level of abstraction is rather low.

2. **Adaptability**

   VoiceXML does not provide mechanisms to store users' preferences and automatically adapt dialogs based on these preferences, or to react to environmental factors. The issue of different user interfaces on different devices does not apply to VoiceXML, since the language focuses on speech-based interfaces that are presented in the course of one telephone call using exactly one device.

3. **Openness**

   The full specification of all language versions of VoiceXML was published as recommendation by the W3C and is available online.

4. **Organizational background**

   VoiceXML was originally developed by the VoiceXML forum[16], a Consortium founded by AT&T, IBM, Lucent and Motorola. The World Wide Web Consortium (W3C) standardized VoiceXML and currently is responsible for further development of the language.

5. **Status**

   The latest version 2.1 [32] was published in 2007. The main goal of the major 2.0 version published in 2004 was to consolidate and standardize various adaptations of the 1.0 version made by third party developers. Version 3.0 is currently available as public draft.

6. **Number of implementations**

   VoiceXML is used by many implementations of automatic telephony systems on the World Wide Web.

7. **Number of supported target platforms**

   VoiceXML is designed solely to support speech-based user interfaces.

2.2.15 Hypertext Markup Language (HTML)

The HyperText Markup Language (HTML) is the markup language used for the specification of web pages. This section deals with version 4 (latest version is 4.01 [41]) since it is the first version that separates semantics and appearance. Except for some minor differences, the analysis also applies to the eXtensible Hypertext Markup Language (XHTML) [36]. HTML is based on the Standard Generalized Markup Language (SGML), while XHTML is based on XML. XML actually is a subset of SGML that adds additional restrictions to the base language [18]. Therefore HTML and XHTML differ only slightly in their syntax, but they share the same language elements and the general concept of their use as hypertext markup language.

[16] `http://www.voicexml.org/`

level of abstraction:	rather low
adaptability:	
accessibility:	no
context-awareness:	no
use-case awareness:	N/A
openness:	high
organizational background:	open standards community
status:	under development
implementations:	many
supported target platforms:	1

Table 2.14: Comparison chart for VoiceXML

Language specification

A HTML document describes the content elements of a web page, but not its appearance. In addition, it may include meta-information that further declares the page's content, its author, the language, keywords, etc.

HTML documents consist of two parts: The `head` element contains the above-mentioned meta-information. For defining the user interaction, the `body` element is more interesting: It contains all the visual elements that shall be presented to the user.

HTML defines several elements that may be used as user interface widgets inside an HTML document. Elements may be nested. HTML includes basic elements for embedding text, images, tables, and other static information. For specifying text layout, special elements are available, e.g. the `h1` through `h6` elements that define text heading on six hierarchical levels. It should be noted however that all those text-specific elements only encode the semantics of a text paragraph, not its exact appearance.

To integrate interactive elements in a web page, HTML provides several mechanisms: The main feature of hypertext that distinguishes it from conventional text documents is linking. HTML allows the definition of hyperlinks that can be used to navigate to another web page. In addition, interactive applications can be created by using the `form` element. Such a form can contain usual HTML markup (e.g., a table for specifying the basic layout, images or text paragraphs) as well as special form elements, including widgets such as buttons, text boxes, lists etc.

Each form should also contain a special element called submit button. The user can fill out all the widgets in a form and then click the submit button to submit the form. For each form, a server-side form handler must be specified. On submitting the form, the data that was input or selected in each form widget is submitted to this server-side handler and further processed there. As a result, the server can deliver a new web page displaying a confirmation, for example.

As another way of incorporating interactivity, client-side scripts may be included in HTML documents. The script is executed when the document is loaded or on any other event, for example when a hyperlink is clicked. HTML elements include event handlers that can be used

to trigger the execution of a method defined in a script that is either defined directly inside the HTML document or in an external script document that is referenced in HTML. In general, script support in HTML is independent of scripting languages. However, scripts are not compiled but interpreted by the web browser used to view the web page, therefore the browser must support the script language in order to execute the script.

As mentioned above, HTML follows the concern of separation of content and appearance and specifies only the semantics of web page contents. The exact appearance of each content element is defined by the web browser, although most browsers allow users to change the default appearance, e.g. by setting a standard font size for text paragraphs. To specify the exact appearance of a web page, web designers may add Cascading Style Sheets (CSS) [12] definitions to HTML documents.

CSS allows to reference HTML elements and set these elements' style attributes. To reference HTML elements from CSS documents, CSS provides selectors that select either all elements on a web page, all elements of a given type, all elements that belong to a given class, or exactly one elements by its ID. IDs can be assigned to HTML elements by using the `id` attribute and must be unique within the HTML document. Similarly, classes can be assigned to HTML elements through the `class` attribute. In contrast to IDs, several elements within one document may be assigned the same class, and one element may be assigned several classes. In addition, CSS styles may be assigned directly to an HTML element by using the `style` attribute, instead of declaring all CSS styles in a separate document.

Analysis

The following enumeration contains a detailed analysis of the Hypertext Markup Language according to the criteria defined in section 2.1.

1. **Level of abstraction**

   HTML was developed for the design of visual web pages and therefore includes only elements representing visual content.

level of abstraction:	low
adaptability:	
accessibility:	through browser
context-awareness:	no
use-case awareness:	through CSS
openness:	high
organizational background:	open standards community
status:	under development
implementations:	many
supported target platforms:	5

Table 2.15: Comparison chart for the Hypertext Markup Language (HTML)

2. **Adaptability**

   HTML itself does not provide support for the automatic adaptation of web pages to user's preferences or to environmental factors. Most web browsers however allow users to set basic layout characteristics, e.g. font size or font family, to best fulfill their needs.

   Concerning the issue of displaying different user interfaces on different devices, CSS supports multiple media types: A HTML document may reference several CSS documents, one for each media type the web page shall support. Each CSS document could style the content elements in a different way or even hide some of the elements on a certain group of devices. According to [12], supported media types are:

   - computer screens
   - handheld devices
   - projectors
   - printers
   - braille tactile feedback devices
   - braille printers
   - speech synthesizers
   - TV systems
   - terminals using a fixed-pitch character grid

3. **Openness**

   The full specification of all HTML language versions (including the current 4.01 and XHTML 1.0 versions) is available for free, published by the W3C.

4. **Organizational background**

   HTML was initially proposed by researchers from the European Organization for Nuclear Research CERN. It was standardized and is currently worked on by the World Wide Web Consortium (W3C), an international community developing open Internet standards.

5. **Status**

   The latest version 4.01 was published 1999, the XHTML 1.0 specification was published in 2002. HTML 5 as successor of both specifications is currently under development.

6. **Number of implementations**

   HTML is very common and widely used since it is the basis for nearly all pages available on the World Wide Web. A variety of different web browsers exist, and most of these browsers use different rendering engines.

7. **Number of supported target platforms**

   Web pages are mainly viewed using web browsers running on desktop computers, although mobile devices become increasingly important as mobile Internet access is becoming more common and various web browsers are already available for mobile platforms. Similarly, an increasing number of television systems supports Internet access and the presentation of web pages through rendering HTML content. In addition, there exist web browsers that process HTML content for output with braille tactile devices and speech synthesizers.

2.3 Classification

The analysis of User Interaction Description Languages conducted in this chapter shows that a variety of such language frameworks have been developed especially during the last ten years. Detailed comparison proves significant distinctions concerning the nature of the analyzed languages. For example, not all of these languages are already used in practice, and some are developed by industrial companies and not freely available without taking into account licensing issues, making them less interesting for use in a research project.

The main distinction, however, regards a UIDL's level of abstraction. This criterion allows to draw conclusions about the intended purpose of a language. Most of the analyzed UIDLs that are used in practice are restricted to either a specific programming framework or to a certain type of platform. On the other end of the spectrum, there are languages which focus at the definition of interactions provided by a service rather than the definition of user interface concepts.

Within the use case of a user interacting with a service through a user interface device, each type of UIDL has its advantages for special applications, while none is able to fulfill all purposes. The detailed analysis of User Interaction Description Languages suggests the distinction of three types of UIDLs based on the following classification scheme:

- **Service interaction description languages**

  This type of description language focuses on interactions from the point of view of the service rather than the user. Such an interaction description contains the technical specification of the communication between service and user interface device. In detail, this contains data types to be exchanged, protocols used for the communication, etc.

  A typical example is WSDL: It allows the definition of functions a web service provides. This includes operations necessary for the user interface device to access these service functions, the data types exchanged between service and user interface device, and the sequence in which messages containing these data types have to be sent in order to access the service's functions. It does not define details about how these operations are presented to the user by the user interface device.

- **User interaction description languages**

  This type of description language provides a translation of technical interaction descriptions to the presentation of these interactions to the user. The focus is on the interaction between user interface device and service (not between user and UI device), but in contrast to the first type of interaction description languages this type specifies the data to be presented to the user or requested for input from the user. All this is specified in a platform- and modality-independent way, in order to allow any type of user interface device to be used.

  Examples for this type of interface description language are UIML, UsiXML, and MARIA XML, although some existing languages that fit in this category constrain the above-mentioned modality-independence by supporting only a predefined set of input/output modalities.

Such User Interaction Description Languages that describe user interaction concepts in an abstract way are in theory applicable to all user interface devices regardless of platform and modality, however in practice the translation to a platform-specific concrete user interface description is a difficult task because of the high abstraction level of most modality-independent user interaction descriptions: Due to the lack of modality-specific information, no detailed information about the actual presentation to the user is available which could be used by the UI framework. User interaction description languages that focus on a predefined set of modalities might simplify this translation problem, however they are not universally applicable to all purposes.

- **User interface description languages**

  This category contains languages that are used to specify the interaction between user and user interface device. Usually this is achieved by using UI elements and therefore either platform- or framework-dependent.

  The first type, platform-dependent user interface description languages, provide specific commands and UI elements for a certain platform of user interface devices. Examples are languages that focus on graphical user interfaces to be operated using mouse and keyboard. One specific example is VoiceXML that allows the specification of voice-based user interfaces.

  The second type, framework-specific user interface description languages, provide a markup language for specifying user interfaces to be used by a specific programming framework. Although their use is restricted to one programming framework, they might be applicable to specifying user interfaces for several target platforms. One example is XAML which was designed for use with the .NET framework for the Microsoft Windows platform. It can be used to specify user interfaces for at least two platforms - desktop computers and mobile devices (smartphones, PDAs) - since Windows and .NET are available for both platforms.

  These languages are widely used in practice and may be applied in a very efficient way to use cases following their intended purpose, but are hardly extendable and applicable to use cases beyond that scope.

Many existing services implementing a sequence of interactions between user and service through a user interface device rely on a hierarchical approach, utilizing all three types of UIDLs: The supported functionality of a service is specified using a service interaction description language, and the operations presented to the user are defined in an abstract way through the use of a user interaction description language. For the actual implementation of user interface devices, this abstract description is transformed to one or several framework-dependent user interface description languages.

For example, MARIA XML follows this approach: It relies on a WSDL description of the underlying service, based on which several user interaction descriptions are created, one for each target modality to be supported. Finally, several concrete user interface descriptions are generated using platform-specific user interface description languages that can be directly executed on the desired user interface devices. Compared to an approach that uses only one

abstract user interaction description valid for all modalities, the use of several abstract user interaction descriptions simplifies the final transformation to framework-specific user interface descriptions.

CHAPTER 3

User interaction patterns

3.1 Problem statement

As stated in section 2.3, a hierarchical approach of an automatic user interface generation system consists of several transformational steps: From a service description to one or several abstract user interaction descriptions to several concrete user interface descriptions. Regarding user involvement, the last step is the most important one: The responsibility of this transformation is to generate user interfaces that appear as accessible to the target user group as possible.

Implementing such an automatic system is an even more challenging task than defining accessible user interfaces manually: There is no way to manually adapt the generated user interfaces if they do not represent the optimal solution for a certain application. Instead, rules for the automatic appliance of UI elements need to be defined that take into account accessibility considerations, in order to generate user interfaces that provide optimal user experience in all use cases.

As part of this thesis, a prototypical implementation of a system for automating the abovementioned final transformation of abstract user interaction descriptions into concrete user interface descriptions has been developed. This prototype should cover two target platforms. In order to reduce complexity, these sample platforms should share the same concept of input/output modalities, while being sufficiently different to require different types of user interaction in certain use cases.

As target platforms, mobile phone (smartphone) environments and tablets were chosen. The operation of most modern smartphones and tablets is quite similar: Both platforms usually provide direct interaction through touch screens. Such a screen is used for both output and input. Additional output channels are built-in loudspeakers that can be used for audio output, as well as the possibility for vibration alerts offered by almost all mobile phones and some tablets.

In all systems, text input is generally achieved through on-screen keyboards that overlay parts of the user interface on demand. In addition, there exist smartphones that provide small physical keyboards. Those keyboards are either located beneath the screen, or inside the body of the device and may be flipped out when needed. Most tablets may be connected with hardware keyboards using Bluetooth or other wireless communication channels. Most tablet and smartphone systems also offer a variety of additional input sensors, such as a microphone, acceleration sensors, one or several camera sensors and an A-GPS system for retrieving the device's current location.

That the two platforms are rather similar is also shown by the fact that the most widely used operating systems for smartphones are also used on tablets, but at the same time not available for desktop or laptop computers. Examples include

- the latest major version of Google Android[1], version 4.0, which is the first version that runs on both smartphones and tablets
- Apple iOS[2] which is used on smartphones (iPhone), tablets (iPad) as well as on the MP3 player series iPod touch

[1] `http://www.android.com`
[2] `http://www.apple.com/ios/`

- the upcoming Microsoft Windows RT, one edition of the Windows 8 operating system[3] which will be the first operating system to be used on mobile phones, tablets, and desktop computers (while the previous versions Windows XP and Windows 7, being among the most successful operating systems for desktop computers, have never been widely adopted and accepted on tablet devices)

While there are several similarities between smartphone and tablet platforms, especially concerning interaction concepts and operating systems as mentioned above, there also exist major distinctions. The most obvious one is the different size of the devices and their different screen sizes. This comes along with different display resolution. Using a larger screen with higher resolution means that more information can be displayed at once which brings about more complex user interfaces. More UI elements on the screen also means the possibility of direct manipulation of several objects on one screen, which requires elaborate user interfaces in order not to overstrain and confuse users.

Another distinction between the two platforms not being so obvious is the different way of use of the two types of devices. A mobile phone is more common to be carried with the user, while the use of a tablet device is restricted to certain environments. In addition, the telephone feature of smartphones offers new methods of use.

For testing and evaluating the prototypical implementation, two simple example AAL services have been defined. The next section describes them in detail. The specification of these services has also been used prior to the implementation in order to identify complex patterns of interaction between user and device and to develop optimal solutions of how to represent them in a user interface, resulting in general rules and guidelines for the automatic generation of user interfaces. This process was conducted in collaboration with potential users and is described in detail later in this chapter.

[3] `http://windows.microsoft.com/en-US/windows-8/release-preview/`

3.2 Example AAL services

This section describes the two AAL services that were used as example in the process of development, implementation and evaluation of the user interface generation system prototype.

Since the focus of this thesis is not on the implementation of back-end services but rather on user interaction and user interface generation, both services were chosen due to the fact that they are not too complex from a technical point of view while at the same time requiring interesting patterns of user interaction.

They are different from each other in their general purpose: The first service allows the user to request information from the system, meaning that the user actively starts the service, inputs some data and is presented with information retrieved based on this data. This might include the option for further interaction, which infers a similar phase of data request and information retrieval. In contrast, the second service is a reminder service that is not actively started by the user but rather on its own at a given point of time, requiring to catch the user's attention in order to convey information.

3.2.1 Doctor's appointment service

The doctor's appointment service fulfills two tasks: First, the user may search for a nearby doctor, based on certain search criteria. Afterwards, it is possible to make an appointment at this doctor's office, by finding the exact appointment date offered by the doctor that fits the patient's schedule best.

This service provides a simple solution to a task common to the target group of elderly people living on their own: The need to find a doctor specialized on a certain field, whose office is easy to reach. In addition, it offers the possibility to directly make an appointment after choosing a doctor. Thus, the service combines two tasks (finding a doctor, for example using a phone book, and making an appointment by calling the doctor's office) that belong together but are usually done individually. The result is that both tasks can be achieved in one step, using only one service and one device.

Technically, a solution that may access a doctor's appointment schedule in order to present possible appointment times to the user and to directly reserve one of these time slots requires real-time communication between the service and the server that stores the doctor's schedule. However, since this communication can be fully carried out by the back-end service, the exact implementational details need not to be specified when concentrating on user interaction on the user interface device, and the communication channel need not to be implemented for testing the acceptance of user interaction concepts. Therefore, questions concerning this back-end communication are not in the scope of this thesis.

Application flow from the user's point of view

In the first phase, the user is requested to input certain search criteria that can be used for finding appropriate doctors. Examples of these search criteria are the field of specialization and

types of insurance accepted by the doctor. The list of all available doctors is then filtered based on these criteria, and all appropriate doctor's offices that are positioned inside a certain perimeter are presented to the user, including distance and address. The user can then choose one of them for continuing with the second phase.

After choosing a doctor, the second phase allows the reservation of an appointment. In a first step, the user is asked for the date of the appointment. This may be done by specifying the favored day, or by specifying a period of time. In addition, the user could be allowed to enter preferences for certain week days, or for a certain period of day time. The user is then presented with an overview of all offered appointments fulfilling the specified criteria, and may either go back one step to change the preferences, or choose one of the offered appointments. After one appointment was chosen, a summary of the doctor's contact details and the chosen date and time is displayed, asking the user for confirmation. After the appointment was confirmed both by the user and the doctor's office, a final confirmation message is displayed to the user.

3.2.2 Medication reminder service

The general purpose of the medication reminder service is to provide elderly people with a tool to help planning the daily task of taking medication. Being forced to take different kinds of medicine at different times of day which is a typical situation for elderly people may be confusing and overstraining for some patients.

Mistakes in taking medication might result in reduced efficiency of certain medicaments, but can as well bring about more dangerous effects. The medication reminder service reduces the user's responsibility in two ways: First, the user does not need to remember the exact time for taking some sort of medicine because the service reminds him or her exactly at the right moment. Second, the service can provide concrete instructions which amount of which medicament to take, such that the patient does not need to remember correlations between daytime and type of medicine. Moreover, the service offers the possibility of notifying relatives or nursing staff if the patient does not confirm the medicine taken after a certain period of time.

Application flow from the user's point of view

As any other service, the medication reminder service needs to be activated by the user once through the main menu. This is followed by a request for some basic settings, including the definition of alarms and the input of contact details of a relative or nursing staff to be used in case of emergency. In a second phase, the service is reduced to a background task that launches reminder messages at a certain time.

During the basic setup, a list of alarms must be provided by the user. An alarm contains a time of day, and a textual description of the task to fulfill at that time (usually being the type and amount of the medicament to take at that time). At startup, a list of all existing alarms (which were entered at the service's previous startup) is shown, but items of this list can be edited or deleted and new alarms may be added.

In addition, the telephone number of a relative or nursing staff to be notified in case of

emergency is requested. Also in this case, the value entered at the last startup is provided as default. All information - alarming times, reminder alerts and emergency telephone number - may be input by a relative, a doctor, nursing staff, or by the patient himself if he feels capable to do so. Especially the ability to set up and control the operation of an AAL service by the user might increase the willingness to use such a service in practice. Therefore the user interface of this part of the software must be as accessible and easily operable for elderly users as all AAL services.

After input of these basic settings, the medication reminder service is executed as background task. On alarming time, the service's visible user interface is re-activated and shows a reminder message on the screen of the user interface device, containing the respective instructions for taking medication. The user is asked to confirm that the medicament has been taken, in which case the service switches back to the invisible mode of a background task. If either the reminder message or the screen showing detailed instructions is not confirmed by the user within a predefined period of time, a relative or nursing staff is notified by the back-end service through the provided telephone number.

3.3 Interaction scenarios

The previous section presented the two prototypical example AAL services in detail, including both their general purpose and behavior and their appearance to the user. The detailed sequence of data requested from the user and information presented to the user was specified in sections 3.2.1 and 3.2.2. However, the exact way of how certain data is requested from or presented to the user depends on the user interface device currently used and therefore can not be predefined by the service.

From a technical point of view, the service must specify the type of data to be requested from or presented to the user. In addition, it may define supplemental semantic information in order to further specify the nature of any data item. How input or output of a certain data item is achieved through the user interface device depends on the devices input/output modality. For example, purely audio-based user interface devices are restricted to other user interaction patterns than devices featuring graphical user interfaces.

Most types of user interface devices will nevertheless provide several options to either request or present a data item of a certain data type. This is the case especially for smartphones and tablet devices, as these combine a variety of different input/output modalities, including for example graphical and tactile (through camera sensors, touch screens and vibration alerts) as well as audio-based (through loudspeakers and microphones) data input and output.

Based on the specified application flow of the two example services, six simple interaction scenarios have been identified that may be realized in several different ways using smartphones and tablet devices. This section lists those interaction scenarios. In addition, three research questions are defined which should be answered by evaluating the given interaction scenarios.

3.3.1 Interaction modes

This section describes all interaction scenarios that were identified in the two example services, including all potential options for implementation, some of which may be more or less intuitive and comfortable to use.

Input of a point in time

As part of the medication reminder service, the user is requested to input the exact time of day when the reminder shall be activated. Focusing on the touch screen offered by smartphones and tablet devices, there are several possibilities for the user to enter a certain point in time:

1. **Input using an on-screen keyboard** that is displayed on the touchscreen (or using any hardware keyboard if provided by the user interface device). In this case, the following input modes can be distinguished:

    - One text box is displayed on the screen, in addition to the standard on-screen keyboard. Using this keyboard, the desired point in time is entered, consisting of digits and a delimiting character.

- One text box is displayed on the screen, in addition to a telephone-style on-screen keyboard which allows only the input of digits and certain delimiting characters. Using this keyboard, the desired point in time is entered, consisting of digits and a delimiting character.
- Two text boxes are displayed on the screen, in addition to the standard on-screen keyboard. Using this keyboard, numbers are entered that encode the desired values for hours and minutes.
- Two text boxes are displayed on the screen, in addition to a telephone-style on-screen keyboard which allows only the input of digits. Using this keyboard, numbers are entered that encode the desired values for hours and minutes.

All of these options need an additional validation routine that checks if the point in time that was entered is valid, meaning that it does not contain any characters other than digits, and that the values for hours and minutes are in the interval $[0, 23]$ and $[0, 59]$ respectively. This validation might take place after the time entered was submitted, returning to the time input screen if any invalid data was detected, or in parallel to the input, allowing the submission only if the input is valid.

2. **Selection from a list of predefined values.** Such an approach does not need to validate the entered data, since only valid values may be chosen. Predefined values can be presented in several ways:

   - One text box is shown on the screen which contains an initial value encoding a certain point in time, in addition to two buttons showing + and - signs that allow the user to increase and decrease the current value.
   - Two text boxes are shown on the screen, containing initial values encoding hours and minutes, in addition to buttons showing + and - signs that allow the user to increase and decrease the values of hours and minutes individually.
   - Two wheels containing all potential values for hours and minutes are simulated on the screen. Finger gestures on the touch screen are used for user interaction, simulating rotating movements of each wheel, thus allowing the user to select one of the available values.

   These approaches can be adapted by defining the set of allowed values for hours and minutes. For example, usual values encoding minutes should be in the interval $[0, 59]$, while in some cases that do not need on-the-minute accuracy values encoding quarters of an hour (0, 15, 30, 45) might be sufficient. This makes it easier for the user to find the desired value, since it decreases the size of the list of potential values.

3. **Direct interaction with a simulated analog clock.** In this case, a clock face is displayed on the screen, in addition to a clock hand that points to an initial value. The hand might be touched and re-positioned using finger wipe gestures on the touch screen. Different approaches are imaginable for switching times between a.m. (hours between 0 and 11) and p.m. (hours between 12 and 23), as well as for distinguishing hours and minutes. For

example, on larger screens two clock faces might be displayed for selecting hours and minutes individually, or one clock face featuring two clock hands might be used.

In addition, several of those methods presented above can be combined. For example, a text box for entering a point in time using an on-screen keyboard might be combined with buttons showing + and - signs to allow the user to easily increase and decrease the current value without using the keyboard. Such combined approaches give the user the flexibility to use those input mechanisms he or she likes best.

Input of telephone numbers

The medication reminder service requires the user to input the telephone number of any person to contact in an emergency case. This can be achieved in the following ways:

1. **Input using an on-screen keyboard or selection from a list**. These traditional input methods can be used and combined in the following ways:

   - One text box is displayed on the screen, in addition to the standard on-screen keyboard. Using this keyboard, the desired phone number is entered, consisting of digits and optionally a + sign.
   - One text box is displayed on the screen, in addition to a telephone-style on-screen keyboard which allows only the input of digits and certain special characters such as the + sign. Using this keyboard, the desired phone number is entered.
   - Several text boxes are displayed on the screen, dividing the phone number into several parts such as country code, city code, and number. Country code and city code may be chosen from lists of potential values (the list of city codes dynamically changing, depending on the currently selected country). The number must be entered using an on-screen keyboard (either the standard one, or a special telephone-like keyboard that allows only the input of digits).

   Similarly to the input of a point in time, all approaches that make use of manual input through an on-screen keyboard (or optional hardware keyboard) need an additional validation routine that checks if the entered phone number is valid, which can be executed either while typing or after submitting the results.

2. **Direct interaction with a dial plate.** This approach simulates an old dial telephone by drawing a dial plate on the screen which can be operated using finger wipe gestures. Digits are entered through the dial plate one by one, an optional text box may show the digits that have already been dialed.

3. **Selection from stored contacts**. Instead of entering a telephone number manually, the user might select one of the contacts stored on the phone.

   This method is only appropriate if the user has already added phone numbers to the phone's contacts-application and actively uses it. Furthermore, it is not suitable for use

cases of entering a new contact's phone number, but only for selecting one existing contact, for example in order to declare an emergency contact, as in the medication reminder service.

This method can be combined with others, thus allowing the user to select an existing number if already stored in the phone, or enter a new one otherwise.

Selection of date and time

The doctor's appointment service requires the selection of one of several given appointment dates. In a first step, the desired period of time is constrained to either an exact date or a range of days. Then the user can choose from a set of given appointment dates that are in the selected period of time. This user interaction scenario in its various forms, as described below, is special to the doctor's appointment service, but can be applied to different use cases in similar form.

1. There exist two methods for the input of a period of time that is required for the interaction scenario's first step:

   - Direct input of a certain date, for example selecting the desired date from three lists that show potential values for day, month, and year, or from a calendar-like view.

     In both cases it is possible to prevent the user from selecting dates that are invalid for a certain use case, for example by prohibiting the selection of weekend days, or by restricting the selectable range of time to the next few months.

   - Narrowing down a period of time through providing search criteria. This can include, for example, options for including or excluding certain week days or selecting start and end dates.

2. The second step focuses on presenting a set of available appointment dates and the selection of one of those.

   - If dates are displayed in a list, they will usually be sorted by date and day time to make it easier to find the desired date.

   - An alternative visualization method is a calendar-like view: Each day is evenly divided in shorter periods of time, for example hours, and these are shown in a list. Available appointment dates are displayed in this list exactly at that position that correlates to the start time.

   - If the user interface device to be used features a calendar application, as almost all modern smartphones and tablet devices, the user's personal schedule can be included in the list of available appointment times. This allows the user to sort out conflicting appointment dates at a glance. Alternatively, appointment times that are in conflict with events in the user's private schedule might even be hidden automatically by the system.

Presentation of a list of data items

This interaction scenario focuses on a list of several data items, each of which represents a complex object rather than a simple data type, implying that the contents of the data items can not be edited directly in the list due to a lack of screen space. The user shall be able to add data items to the list as well as edit and delete existing items. One example is the list of alarms presented at the start of the medication reminder service: Each item consists of a time of day and a textual description, and the user may edit, delete, and add items.

If it is not possible to display and edit all contents of each data item while viewing the whole list, either because the screen being to small or the amount of data being to large, it is necessary to provide separate screens for each data item, showing all content of the current data item in detail, including the possibility to edit this content. There are two methods for interaction with a list, including such a detail view:

1. The individual detail screens for each data item are shown in sequence, allowing the user to inspect and, if desired, edit the content of each data item in the correct order. Items may be deleted, in which case the detail screen of the subsequent item is shown. In addition, after the last item in the list a separate screen for a new list item is shown. The user may either fill out the empty screen in order to add a new data item to the list, or skip this step.

2. Alternatively, the most important data of each list item may be identified and used for an overview screen. In this case, two individual screens are provided that are shown alternately: One screen shows the list, including the most important information of each data item, the other screen shows all content of one data item in detail, including the possibility to edit this content.

   In the list view, each data item can be selected (on touch screens, for example, by tapping on one item), in which case the detail view for this data item is shown, allowing the user to inspect or edit all content of the item. On confirmation or cancel, the list view screen is brought back into view.

Presentation of location-based data

The presentation of information which contains location-based data, for example street addresses or exact coordinates, is an example for a user interaction that can be enhanced using some of the possibilities offered by modern smartphones and tablet devices. The doctor's appointment service, for example, might make use of location-based data, since it must present a collection of doctor's offices to the user. In practice, different approaches can be used, some of which may be combined, that make use of the various sensors and input/output devices provided by smartphones and tablets, as well as third-party software solutions:

1. Due to the widespread use of location retrieval sensors in mobile devices, the current position of a device (and therefore its user) can be identified quite accurately. Most systems use GPS (Global Positioning System) to determine the current position, mobile phone or

Internet devices can additionally retrieve the device's current position inside the mobile phone network by identifying the nearest transmitting stations.

If the general information to be displayed to the user includes GPS coordinates, it is not useful to simply display these coordinates in most cases, because they are not interpretable by average users. Instead, the information can be enhanced by interpreting the given coordinates in relation to the device's current location, and present the user with the derived data, for example the distance of a certain point of interest to the current position.

2. In case a street address is available in the given data set rather than exact coordinates, existing third-party geocoding solutions can be used to retrieve the exact location of the given address. Examples of such services include Google geocoding API[4] and Yahoo Maps geocoding API[5]. The retrieved data can then be processed as described above.

3. Third-party map services offer a solution for alternative visualization of location-based data. The location of any given point of interest can be visualized on different kinds of street maps, for example in relation to the user's current position. Touch screens offer a comfortable solution for interacting with such a map visualization. Several of those map services are available, among them the well-known Google Maps[6] and Bing Maps[7] solutions.

4. Third-party services can even help in the creation of more advanced user interaction scenarios, for example the integration of automatic route planner services that allow the user to find the fastest way from the current position to the location of any given point of interest.

Alarms and reminders

One of the key features of the medication reminder service is a background service that awakes at a certain point in time in order to remind the user to take some medicine. Self-activating alarm and reminder services are an important use case of mobile devices, and crucial to AAL systems. The main difference compared to other typical user interaction scenarios concerning mobile devices is that the interaction is initiated by the device rather than by the user, with the aim of drawing the user's attention to the device and then present some important information to the user.

An alarm must therefore be sufficiently alerting to catch the user's attention, while on the other hand it must not be annoying - the second case might lead users to switch off the service completely. Several aspects must be taken into account in order to generate effective reminder interactions on mobile devices:

[4] `https://developers.google.com/maps/documentation/geocoding/`
[5] `http://developer.yahoo.com/maps/rest/V1/geocode.html`
[6] `https://developers.google.com/maps/`
[7] `http://www.microsoft.com/maps/developers`

1. **Alerts**

   The first step of a reminder-style user interaction is an alert that aims at drawing the user's attention to the device. The following types of alerts are imaginable on mobile devices:

   - Auditory alerts are most common on smartphones and tablet devices. For example, incoming phone calls and messages are usually announced by playing a sound using the device's internal loudspeaker. Differentiation is possible through the type of sound, its volume and duration.
   - Vibration alerts can be used alternatively or supplemental to auditory alerts.
   - Visual alerts are useful if the device is positioned near to the user and visible to him or her. Displaying some reminder on the screen might not be sufficient since the screen may be turned off or too dark to see from larger distances. Alternatively, turning on the screen automatically might catch the user's attention. In addition, many mobile devices feature LED lights nearby the screen which can be used for alerts by glowing or flashing in certain colors.

2. **Message display**

   After the alert that draws the user's attention to the device, a reminder service's main aim is to present some important message to the user.

   - If some text message or any other information is to be displayed on the screen, the simplest solution is to interrupt any currently running application, clear the screen and fill it with the information to be presented.
   - An alternative to a separate screen for the reminder is the use of some kind of message box which is displayed on top of the currently executed application. As in the first case, the current application is also interrupted, but the more lightweight message box that does not occupy the full screen but leaves parts of the current application shine trough emphasizes that only a short interruption is taking place.
   - If the reminder message shall still be presented using the screen but without interrupting any application that is currently running, a small area on the top or bottom of the screen can be dedicated to the reminder message, leaving the remaining screen area to other applications. This approach notifies the user while at the same time not interrupting any other application he or she might be currently working with.
   - Other approaches are possible alternatively or supplemental to displaying a reminder message on the screen, for example using the built-in loudspeakers to play an audio message.

3. **Life cycle**

   This aspect focuses on the end of a reminder action. Several decisions need to be taken, answering the following questions:

- How long shall the auditory/visual alerts be active? Shall any alert sound be played once or repeatedly?
- Is a confirmation by the user requested, or is the reminder message hidden automatically after a certain period of time?
- If the user must confirm the reminder message, what happens if this confirmation does not occur within a certain period of time?

3.3.2 Questions for evaluation

As basis for the implementation of a prototypical version of a user interface generation system, user tests were conducted in order to evaluate how potential users of the system can interact with the system in the best way, and which technical requirements must be fulfilled for implementing these optimal interactions. The general aim of this evaluation was to answer the following three questions:

1. *For each of the user interaction scenarios identified above, which of the suggested solutions is most suitable for the target group of the system?* How do potential users interact with the various implementation methods for each interaction scenario, which method is most intuitive and should therefore be used by the automatic user interface generation system to create a user interface resembling the respective interaction?

2. *How much information should be shared between back-end service and user interface device?* For example, if a service requires the user to input a telephone number it is sufficient to ask the user interface device to request the user to enter data of type number. However, the service could additionally inform the user interface device that the requested number will be used as telephone number in the course of the service execution. In this case, a more appropriate user interface can be generated (for example, the user could be given the possibility to select the number from his contacts in addition to manually entering a number).

3. *Are potential users willing to allow services to access their private data?* For example, the interaction scenario of entering a telephone number can be more comfortable if the user is allowed to select a number from a list of personal contacts instead of entering it manually. The necessary access to private data should only be granted to the AAL service application, if user interaction scenarios which make use of this private data are appreciated by potential users in the course of user tests.

The following section summarizes the user tests in detail, including the detailed questions that were studied for each of the user interaction scenarios listed above.

3.4 Test preparations

In the course of the user tests, persons belonging to the target group of the user interface generation system should be able to test the interaction with the two example services defined in section 3.2. In order to evaluate which of the interaction modes listed in section 3.3.1 is the best solution, for each of the user interaction scenarios several test cases were defined which the users should handle and compare.

To make the interaction as realistic and intuitive as possible, mockups were created as part of the preparation for the user tests. This section discusses the way the exact test procedure was defined, the different types of mockups that were created, and finally lists the user interaction modes that were chosen to be used in the tests, including the questions to be answered in the course of the tests.

3.4.1 Test procedure

Several different test procedures were discussed prior to the user tests. The initial plan was to show test users each of the two example services at once, letting them step through the whole service from the first screen to the last. This procedure would have been repeated several times, each time replacing one interaction mode by another one, in order to process multiple implementations of all of the defined user interaction scenarios. Both services contain multiple of those interaction scenarios: two of the scenarios belong to the doctor's appointment service, the other four scenarios to the medication reminder service. Again, for each scenario several interaction modes were chosen to be tested.

To combine all available interaction modes of one scenario with all interaction modes of all the other scenarios that are present in each service, many iterations would have been necessary, implying the risk of overstraining test users. In addition, this test procedure would have taken a long time to conduct.

To overcome the drawbacks of the test scenario outlined above, the test procedure was slightly adapted. Interaction scenarios that do not depend on their exact setting in one of the example services have been extracted, in order to be discussed with test users individually prior to presenting the full example services. This concerns the scenarios *input of a point in time*, *input of telephone numbers*, and *presentation of location-based data*. Test users should be presented various implementations of these interaction scenarios, followed by a discussion with the aim of finding the optimal solution for each scenario.

In the following user test of the full services, only these optimal interaction modes are used, reducing the amount of necessary iterations to the number of interaction modes for the *presentation of a list of data items* (medication reminder service) and *selection of date and time* (doctor's appointment service) interactions, respectively.

The *alarms and reminders* user interaction can be considered a special case: As for all the other scenarios, multiple modes of implementing alarms and reminders were to be tested. However, in contrast to most of the other interaction modes, alarms are nearly independent of the service's application flow, since they are launched by a background services at fixed points in

time. Therefore the different versions were initially planned to be executed one after the other, without requiring to iterate trough the full medication reminder service's setup phase each time.

The experience gained in the course of the first user test showed that the *alarms and reminders* interaction is largely independent of the medication reminder setup, and that the connection between the setup phase and the alarms can be clarified more effectively by explaining the context in detail than by iterating through the whole service's application flow. Therefore it was also extracted and from the second user test on executed totally on its own, just like the *input of a point in time*, *input of telephone numbers*, and *presentation of location-based data* interactions.

3.4.2 Mockups

The process of mockup creation was done in several iterations. In a first step, the application flow of the two example services as defined in sections 3.2.1 and 3.2.2 was summarized. For each screen, the necessary user interface widgets as well as the possible user interaction and tasks to be performed were defined in detail.

In addition, the sequence of those screens was specified, resulting in a detailed schedule of stepping through the full services. Each screen, except for the first and last one in each service, has one predecessor and one successor. For some of the screens - basically the ones that represent the user interaction scenarios listed in section 3.3.1 - multiple versions exist, resulting in branches in the schedule.

The aim of the second phase was to design the exact look of each of the screens defined in the first step. The contents of each screen were sketched using a mockup sketching tool. These user interface mockups should be detailed enough to show potential users of the system how they would be able to interact with the service. While designing the mockups, special focus was on those screens that represent the different versions of the user interaction scenarios, in order to show the differences between the interaction modes.

Figure 3.1 shows the sketched mockups for the screen that asks the user to enter a descriptive text and choose point in time in the course of creating a new medication reminder. For this screen, six interaction modes have been chosen as test cases, as described in detail in the next section. Therefore six separate mockup sketches were created, each showing one interaction mode for the interaction scenario *input of a point in time*.

This first generation of user interface mockups showed the application flow of the example services, but they were not considered suitable for user tests, because of their static nature. The example services contain user interface actions that are more likely to be understood by average users if presented in motion rather than using still images. An example is the transition of the last screen of the medication reminder service's setup phase and its further execution as background service, indicated only by a small icon in the status bar.

Another reason for not using the sketched mockups in the course of the user tests was the fact that an important aspect of the tests was to analyze user's interaction with certain user interface widgets, for example using finger gestures on the touch screen, which is not possible using still images that do not react to user actions.

Figure 3.1: Sketched mockups of the *input of a point in time* interaction scenario

Therefore, a second generation of mockups was created in the form of executable programs that can be run on mobile phones. As explained in the test procedure description above, for each of the interaction scenarios to be tested (except for the *presentation of a list of data items* interaction) such a program was developed. These contain just a few screens, all showing the same interaction scenario but each using another interaction mode consisting of different user interface widgets.

These screens are arranged in a way that the user can use finger gestures to switch between screens, wiping to the left/right switches to the previous/next screen. Each screen also contains a button that can be used to switch to the next screen after the current interaction mode was successfully completed. Figure 3.2 shows the screens for the *input of a point in time* user interaction.

In addition, two separate programs were developed showing the full application flow of the two example services. Before starting the visible service application interface, they show a setup screen to be operated by the supervisor rather than by the test user. For all of the user interactions occurring in the particular example service, the setup screen allows the selection of one interaction mode. In this screen, the test supervisor should choose those interaction modes that were identified as best suited by the test user. During the example service, only these interaction modes will be used.

The mockup programs were developed using the Google Android SDK[8], due to the free availability of the Android SDK, the availability of a testing device running Android to be used for the user tests, and the high market penetration of Android devices. The use of a widely used platform was considered important for providing as many potential test users as possible with a familiar platform and interface. For the implementation, Android's standard user interface widgets, styles and themes were used, in order to provide a familiar look-and-feel for those test users that are already familiar with Android devices.

3.4.3 Test cases

Based on the user interaction scenarios defined in section 3.3.1, six test cases were developed that should be handled by test users. For each scenario, several of the suggested interaction modes were designed and implemented in detail. Test users were asked to interact with each of the different implementations and discuss which type they liked best and how interactions can be improved to fulfill the target group's needs.

To avoid confusion and overstraining of test users, not all suggested interaction modes were chosen, since some of them are rather similar to each other. The final selection of interaction modes was based on different criteria: First, the solution that is technically simplest to implement should be present in the test in all cases. The same applies to the interaction mode that represents the currently common solution for the given interaction scenario. In addition, two interaction modes to be tested should not be too similar, the difference should be apparent to an average user. The interaction modes that were chosen are listed in this section, including the questions to be answered in the course of the user tests.

[8] `http://developer.android.com/sdk/`

Input of a point in time

The following modes were chosen to be compared, as shown in figure 3.2:

1. One text box is displayed on the screen, in addition to the standard on-screen keyboard. Using this keyboard, the desired point in time is entered, consisting of digits and a delimiting character. Input validation is done while typing, if the content of the text box does not represent a valid point in time a small exclamation mark is shown next to the text box.

2. Two text boxes are displayed on the screen, in addition to a telephone-style on-screen keyboard which allows only the input of digits. Using this keyboard, numbers are entered that encode the desired values for hours and minutes. Also in this case, a small icon showing an exclamation mark is used as validation check.

3. Two text boxes are shown on the screen, containing initial values encoding hours and minutes, in addition to buttons showing + and - signs that allow the user to increase and decrease the values of hours and minutes step by step.

4. Two wheels containing all potential values for hours and minutes are simulated on the screen. Values can be selected by flipping the finger on the touch screen which simulates rotating movements of the wheels.

5. Two wheels which are also operated using finger gestures are shown on the screen, but the second wheel contains only the values 0, 15, 30, and 45.

6. A clock face is displayed on the screen, in addition to two clock hands that points to initial values. The hands might be touched and re-positioned using finger wipe gestures on the touch screen. Since only one hand can be moved at a time, the second one is rotated automatically as if adjusting a real clock. If the hour hand crosses the twelfth hour, the time is switched between morning and afternoon. The clock face always shows numbers of 0 to 11 or 12 to 23 to illustrate this differentiation. At the bottom of the screen, the currently selected point in time is displayed in numbers.

The general aim of this test was to figure out if direct input through keyboards, selection from a list of potential values, manipulation with + and - buttons, or the direct interaction with clock hands is considered best by test users. In addition, methods 1 and 2 were chosen to analyze if users can differentiate between the standard keyboard and a keyboard that shows only numbers. Methods 4 and 5 were chosen to ask users if the selection of a smaller amount of possible values if more comfortable, and if a potential improvement of input comfort is worth the reduced accuracy.

Another reason for the importance of comparing methods 4 and 5 is that they are implemented in different ways from a technical point of view: To reduce the number of allowed values, the exact purpose of the interaction scenario must be known to the user interface device. Selecting a value for minutes in an interval of 15 minutes might be accurate enough for a medication reminder, while other use cases require the exact definition of hours and minutes. In this case, semantic information must be provided by the back-end service in addition to the raw data type to be requested from the user.

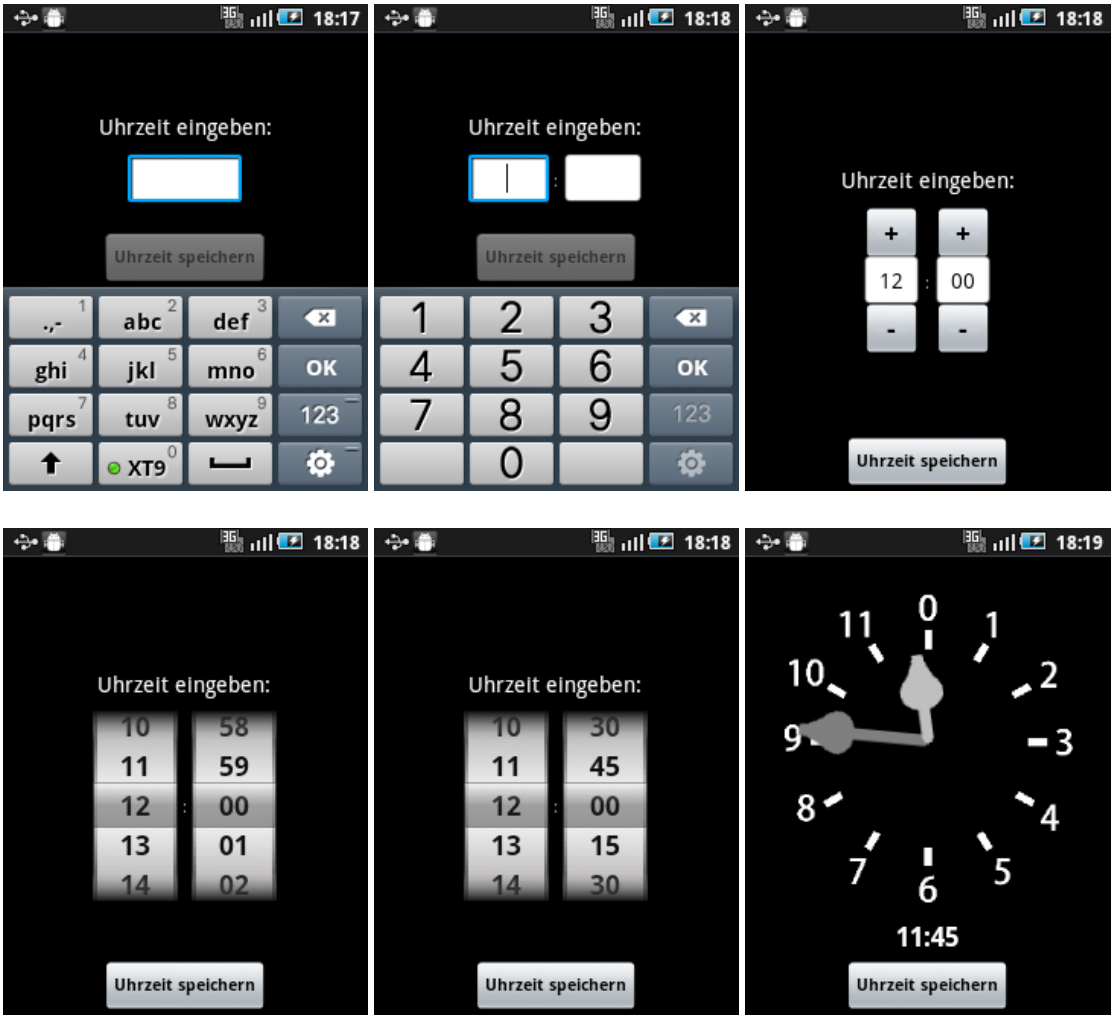


Figure 3.2: Screenshots of the final mockups representing all modes of the *input of a point in time* interaction scenario

Input of telephone numbers

The following modes were chosen to be compared, as shown in figure 3.3:

1. One text box is displayed on the screen, in addition to the standard on-screen keyboard. Using this keyboard, the desired phone number is entered, consisting of digits and optionally a + sign. Input validation is done while typing, if the content of the text box does not represent a valid phone number a small exclamation mark is shown next to the text box.

2. One text box is displayed on the screen, in addition to a telephone-style on-screen keyboard which allows only the input of digits and certain special characters such as the +

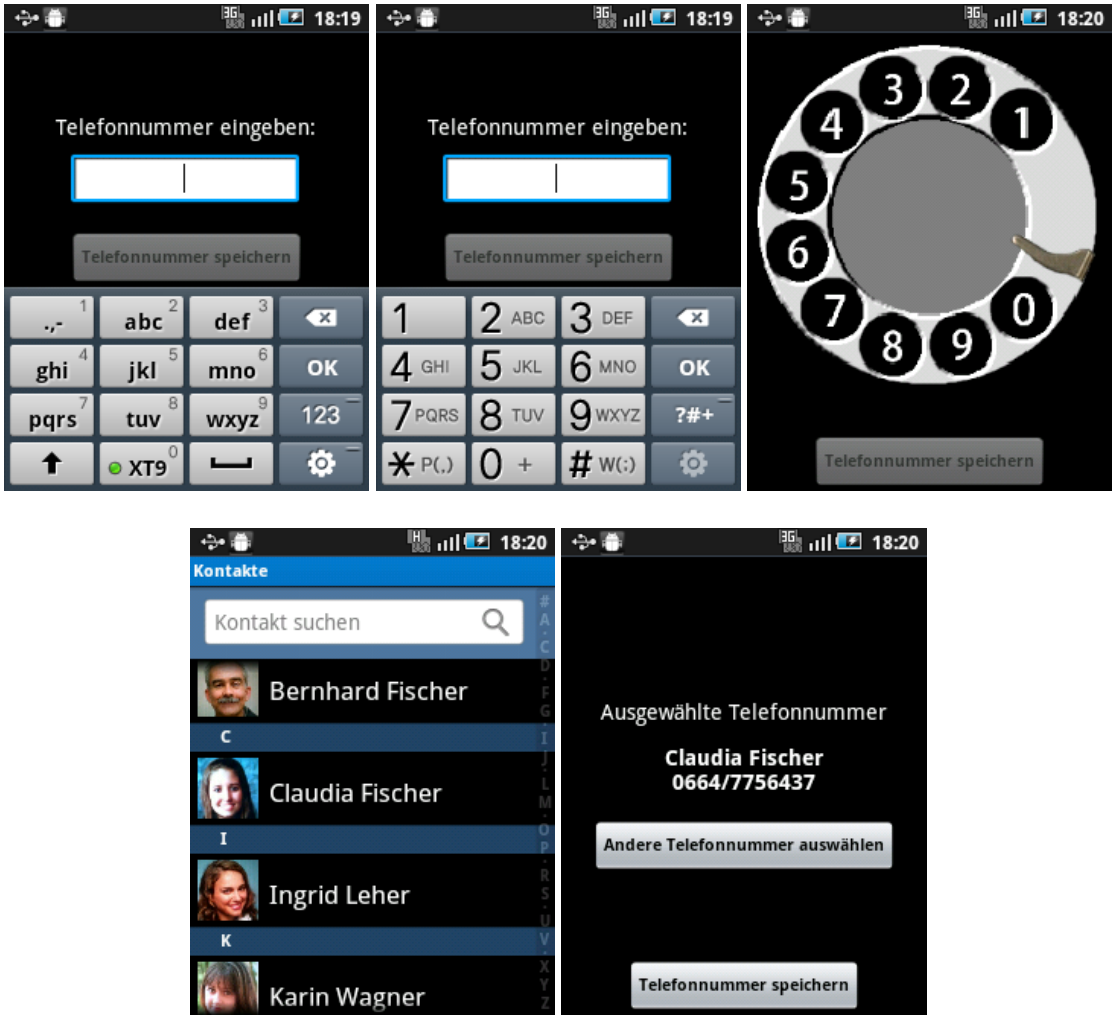


Figure 3.3: Screenshots of the final mockups representing all modes of the *input of telephone numbers* interaction scenario (the last two belonging to the same mode and displayed in sequence)

sign. Using this keyboard, the desired phone number is entered. Also in this case, a small icon showing an exclamation mark is used as validation check.

3. An old dial telephone is simulated by drawing a dial plate on the screen which can be operated using finger wipe gestures. Digits are entered through the dial plate one by one, a short sound message indicates that a digit has been accepted. An additional text box at the bottom of the screen shows the digits that have already been dialed.

4. Instead of entering a telephone number manually, the user might select one of the contacts stored on the phone. After the selection, the contact's name and phone number are shown on the screen, allowing to change the selection and to finally confirm it.

Similar to the *input of a point in time* scenario, this test should show whether users prefer direct interaction with a simulated clock to the simple input through a keyboard. In addition, it should be analyzed if the selection from a list of personal contacts if approved by test users, since such an approach would require access to the users' personal data.

Selection of date and time

The following combinations of interaction modes for date input and appointment selection were chosen as test cases, as shown in figure 3.4:

1. In a first screen, day and month may be selected from two lists. Only dates in the next three months are provided. After choosing a date, the week day is added automatically to make it easier for the user to find appropriate dates.

   After confirming the selection, in a second screen a list of available appointment times is shown. Each list item contains the estimated start and end time of a doctor's appointment. The user can select an appointment by tapping on the respective list item, or use a hardware button to go back one step and change the selected date if no appropriate appointment can be found.

2. The first screen contains widgets for providing search criteria instead of selecting a fixed date: Check boxes can be used to include or exclude certain week days, and sliders are provided for selecting the approximate period of time (allowing the selection of any period between the current date and the following three months) and the day time in which the appointment should be.

   After confirming the search criteria, a list of available appointments is shown, similar to the first test case. Each list item contains the date in addition to start and end time, since appointments might be on different days in this case. The user can select one appointment from the list or navigate back to change the given search criteria.

3. In a first screen, the date can be selected from a calendar-like view. A grid showing all days of a full month is displayed, buttons showing arrows are used to switch between the current and the following two months. Dates that are in the past are shown in a darker color and may not be selected. The same applies to days on which no appointments are available, such as weekends. If the user tries to choose such a day, a message is displayed explaining why the selection is not valid.

   After selecting a valid date by tapping on it, a second screen is shown containing the schedule for the chosen date. The day is divided in shorter periods of one hour which are shown in a list. Available doctor's appointment dates are displayed in this list exactly at that position that correlates to the start time, accompanied by the user's personal appointments. Appointments are visualized in different colors in order to distinguish between private and doctor's appointments. The latter might be tapped on to select and book an appointment.

Figure 3.4: Screenshots of the final mockups representing all modes of the *selection of date and time* interaction scenario, each pair shown in a row belonging to the same test combination and displayed in sequence

This test allows users to compare the direct selection of a date to the method of limiting the range of potential dates through search criteria. The aim is to figure out which method is preferred in general, and how it can be implemented best. The latter focuses on the comparison of the selection from lists to the selection from a calendar-like view, and a discussion about which search criteria are considered useful and how they can be displayed in the best way. In addition, the test shall analyze whether users find the integration of their personal schedule's data comfortable enough to allow an application to access this data.

Presentation of a list of data items

The two methods for interacting with a list including detail screens that were listed in section 3.3.1 were implemented as part of the medication reminder service to be analyzed in the course of the user tests. In the setup phase of the example service, all existing reminders shall be shown, allowing the user to edit, delete, and add reminders. This can be achieved in two ways:

1. Only a detail screen is available, showing a text box for entering the reminder's caption, and widgets for defining the point in time of displaying the reminder, according to the different methods of the *input of a point in time* test case. In addition, two buttons are shown at the bottom of the screen, one for confirming the current reminder's caption and time, and the other one for removing the current reminder from the list. Several instances of this screen are shown in sequence, one for each existing reminder, with the confirmation button switching to the next reminder.

   After the last reminder's details were displayed, an additional screen of the same type is used for adding a new reminder. In this screen, the caption of the second button is changed to indicate that is does not remove an existing reminder, as in the previous screens, but skip the addition of a new reminder. If a new reminder was added, the screen is shown again to allow the user to add another one. If not, the service's application flow is continued with the input of a phone number.

2. A list showing all reminders is displayed. Each list item shows the respective reminder's time and the first few characters of its caption. On selecting one list item, the detail screen for this particular reminder is shown, containing one text box for entering the reminder's caption, and widgets for defining the point in time of displaying the reminder, according to the different methods of the *input of a point in time* test case. Two buttons are displayed at the bottom of the screen, one for confirming the current reminder's caption and time, and the other one for removing the current reminder from the list. Both buttons close the detail screen and return to the list view.

The aim of this test is to figure out if users feel comfortable with switching between list view and detail view as required by method 2, and if this approach is intuitive to interact with, meaning that users are able to select reminders, edit or delete them, and return to the list view, without detailed explanation by the supervisor being necessary. In contrast, method 1 needs more screens to be clicked through in order to step through the full setup phase, but it might be easier and more intuitive to use.

Presentation of location-based data

Two methods of displaying location-based data were chosen to be tested, as shown in figure 3.5:

1. A list of available doctors is shown, including name, address, and distance from the user's current location. By tipping on one of the list items, this doctor is selected.

2. A map showing the neighborhood of the user's current location is displayed. Colored pins in this map represent the user's location, and the addresses of the five nearest doctor's offices. By tapping on one of the pins, details are displayed. A doctor can be chosen by tapping on a special button at the bottom of the screen.

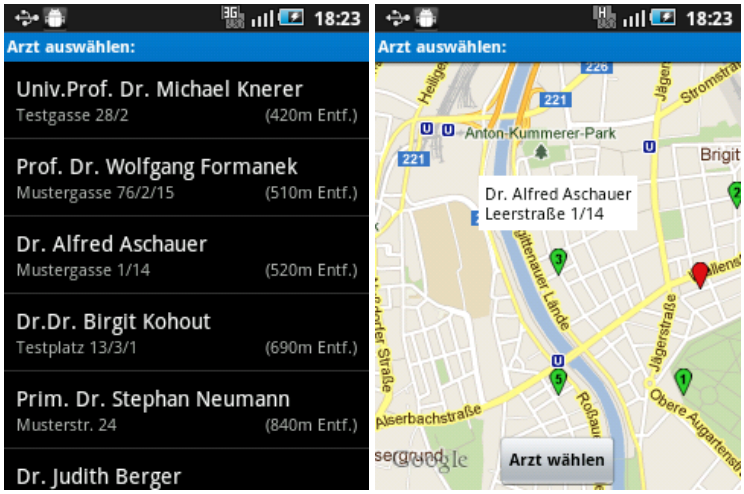


Figure 3.5: Screenshots of the final mockups representing the two modes of the *presentation of location-based data* interaction scenario, displayed after the setup screen

In the course of the user tests, the screens representing these two interaction methods were presented in the context of the doctor's appointment service, and combined with a setup screen to be shown as the first screen in the test. This screen contained several widgets for providing search criteria such as types of insurance accepted by the doctor's office to be found, and week days on which the doctor's appointment may be. Figure 3.6 shows this setup screen. Afterwards, one of the two test screens was shown.

The aim of the first method is to figure out if the distance displayed as part of each list item is considered useful by test users, and which other geographical data would be practical to be shown. The second method is used in order to analyze how intuitive the map view is to interact with, and whether it is a valuable alternative to the list view. Finally, the test users should discuss which prerequisites must be fulfilled to optimally include a similar list view in the use case of finding nearby doctor's offices.

Figure 3.6: Screenshot of the final mockup representing the setup screen of the *presentation of location-based data* interaction scenario

Alarms and reminders

To test which alerting mechanisms are most effective, the following three test cases were incorporated in the course of the medication reminder example service:

1. A short sound is played, similar to the reminder sounds typically used by mobile phones for indicating incoming messages. On the screen, a message box is displayed on top of the currently active application, containing a text message telling the user which medication to take. When tapping on an additional confirmation button, the message box is hidden and the execution of any currently running application is continued.

2. A sound message is played continuously, and the full touch screen is filled with a new window containing a medication reminder text message and a confirmation button, interrupting any currently running application. When tapping on this button, the window is closed, returning to the previously shown application, and playback of the sound is stopped.

3. No audio-message is played, but the device vibrates. At the top border of the screen, the medication reminder text message is displayed, accompanied by a confirmation button that hides the message. The main difference to the other two test cases is that the currently executing application is not interrupted, only the screen space that is available for this application is reduced. The user can continue working with other applications while still having the medication reminder's message in his field of view.

The appearance of these test cases on the screen are shown in figure 3.7. They were executed in a row, with interruptions of 30 seconds in between, allowing test users to directly compare all aspects.

The following questions should be discussed with test users in the course of this test:

- Is a continuously played sound reminder considered annoying or important in order to draw the user's attention to the device?
- Is the vibration alert a useful addition to audio-based and other alerts?
- Which other types of alerts are imaginable? In particular, are visual alerts such as flashing display or illuminated LED lights considered useful as alternative or addition to audio-based alerts with the aim of drawing the user's attention to the device?
- Should medication reminders be allowed to interrupt the execution of other applications?

Figure 3.7: Screenshots of the final mockups representing all tested reminder methods

3.5 User tests

Workshops were performed with eight test persons belonging to the target audience of the AAL example services. The target group of those user tests was defined as containing retired persons older than 55 years who do not suffer from cognitive defects such as dementia or Alzheimer's disease.

The workshops were performed in March and April 2012. The test persons were between 65 and 86 years old. Three male and five female persons participated in the tests. Although not defined as required for taking part in the workshops, all of the test persons lived on their own or with their partner or family rather than in nursing homes for elderly people. The workshops took place in the test persons' homes, and in one case in the premises of a support group for training the cognitive abilities of elderly people. They were performed individually or in groups of two persons.

The aim of the research project and the test procedure of the workshops were explained in detail to all potential test persons. Prior to starting the user tests, they were asked if they were willing to participate. They were informed that participation would be on a voluntary basis, and asked for their compliance to video- and audio record of the workshops.

Each workshop consisted of two phases: First, a short oral interview was performed concerning the test person's background information, in order to figure out how to evaluate the expected results of the second phase. This phase consisted of practical user tests that allowed test users to handle the example service mockups on a real smartphone device.

3.5.1 Interviews

The interview performed initially at the start of the user workshops had two aims: Primarily, it was used to get to know the test persons, their backgrounds and especially their habits concerning technical devices. In addition, as a relaxed conversation that was not recorded it helped the test persons to become acquainted to the test situation. Instead of recording the interview on video or audio, the answers were written down manually.

Apart from personal data such as age and gender, the questions concerned the test persons' attitude towards electronic devices, especially focusing on mobile phones, and typical habits in everyday-use of these devices. The detailed questions were as follows:

1. Do you own a mobile phone?

   If yes, the following questions were included:

   a) Which type of mobile phone is it?

      (This question did not aim at the exact model and brand, but rather at a classification as smartphone with touch screen, classic mobile phone with numbered keyboard, special phone for older users, etc.)

   b) When did you get your first mobile phone, and for which primary reason?

c) What do you typically use the phone for, apart from simple phone calls, and how often?

d) If you are not in your home, do you usually carry the mobile phone with you?

e) If you are at home, is there a common place where the mobile phone is situated, or do you carry it with you?

f) Is the mobile phone always turned on? If not, in which situations and how often do you turn it off?

2. Apart from mobile phones, which other electronic devices do you use?

3. For each of these devices, how often and for which purpose do you use it?

One aim of these questions was to provide an informative basis about the target groups' general attitude to mobile phones, although of course not being representative due to the small number of participants, in order to estimate if mobile devices such as smartphones and tablets are beneficial to be used as platform for AAL services at all. On the other hand, the interview questions provided important information for the interpretation of the interaction tests that were performed in the second phase of the workshops.

For example, a test person who has never used a mobile phone may interact with the example service mockups in another way than a user who has owned a smartphone similar to the one used in the workshop for several years. Both cases are informative to analyze, but must be interpreted in different ways. Therefore, questions 1, 1a, and 1b were used to figure out if a test user has experience in the use of mobile phones in general and mobile devices with touch screen in general.

Similarly, question 1c aimed at a test person's experience in using mobile devices beyond the simple use case of dialing a telephone number and answering incoming phone calls. In addition, a user's willingness to use mobile phones for purposes not related to phone calls might relate to his willingness to use it as interface device for AAL services.

The purpose of questions 1d, 1e, and 1f was to figure out the role of mobile phones in a test person's everyday life. If, for example, a user owns a mobile phone but rather uses it like a landline phone that always stays at home or is even turned off most of the time, it is not the right device to use as interface to AAL services. This applies especially to the medication reminder example service which requires a user interface device that the user pays attention to most of the time.

Finally, questions 2 and 3 were included in the interview due to two reasons:

- Persons who own similar devices such as personal computer, laptop computer, tablet device, etc. are generally more familiar with the use of a typical smartphone user interface, leading to other interpretations of the results of the interaction tests performed in the workshop's second phase.

- If the answers to these two questions show that a test person owns another device, e.g. a TV set, that is used much more often than a mobile phone, the smartphone platform to be evaluated in the course of the workshop is probably not the right device to be used as AAL user interface device.

3.5.2 Interaction tests

These tests were performed using the mockups presented in section 3.4.2 with the aim of analyzing how users handle different interaction scenarios. As explained, the mockups were implemented using the Android SDK and presented using a Samsung GT-I5500 smartphone running Android 2.2. This device basically features a touch screen, the hardware buttons typical for Android devices (including the button for going back one step which is the only one used in the course of the user tests), and various sensors and input and output components, but no hardware keyboard. Both screen brightness and audio volume were set to the maximum value.

The interaction tests were audio- and video-recorded for easier evaluation. Test users were asked to verbalize their thoughts during the interaction with the different test cases, following the think-aloud method [24]. After each test case, a conversation with the test supervisor followed to discuss the main questions to be answered by each test case. The users' interactions with the test device were also video-recorded to be able to reconstruct their interactions with the touch screen later on.

Before starting the interaction tests, test users were presented with typical Android user interface screens to get familiar with the look-and-feel and to practice the interaction with a touch screen interface. For this, the Android settings menu and the screen for adjusting display settings were used. The first screen features a simple list of several items one of which may be chosen, allowing the explanation of finger wipe gestures for list scrolling and simple tap gestures for selecting list items. The second screen shows a simple check box that can be selected and de-selected, and a message box containing a slider widget and two command buttons. The purpose of this screen was to present typical user interface widgets rather than practicing direct interactions.

After the preparations listed above, the test cases defined in section 3.4.3 were performed. Following the test procedure explained in section 3.4.1, the tests containing exactly one interaction scenario were performed first. Although being executed individually, for each test case the detailed context and setting of the example service it was specified for was explained. After iterating through all interaction modes to be tested for a particular scenario, the test user was asked to discuss which interaction mode he considered most useful, and how the different implementations could be improved, and to answer the specialized questions defined for each test case.

Afterwards, the two example service implementations were started for analyzing the *alarms and reminders*, *selection of date and time*, and *presentation of a list of data items* interaction scenarios. As mentioned in section 3.4.1, after the first user workshop the *alarms and reminders* interaction was tested individually. In addition, the experience gained in the first two workshops showed that the *presentation of a list of data items* interaction scenario could not be tested in a reasonable way, as explained in detail in the next section. Those two findings made the execution of the medication reminder service unnecessary, therefore it was skipped in the following workshops.

In contrast, the full doctor's appointment example service was included in all workshops, due to being necessary for analyzing the *selection of date and time* interaction scenario that could not be tested in a reasonable way when excluded from the service context. However, the first

phase of the doctor's appointment service - the selection of a particular doctor - was performed by the test supervisor, since the test user had already intensively interacted with this phase in the course of the individual test case representing the *presentation of location-based data* scenario.

3.6 Results & Conclusions

This section presents and discusses the answers given in the course of the user interviews as well as the observations made during the interaction tests and the subsequent analysis. Finally, conclusions are drawn for the implementation of the prototypical version of a user interface generation system.

3.6.1 Interviews

Concerning mobile phone usage, the group of test persons proved to be quite homogeneous, most of the given answers were similar. Several surprising aspects were figured out that are in contrast to the expectations stated prior to the workshops:

- All of the test persons stated to own and regularly use a mobile phone, although none of the mentioned devices are smartphones featuring touch screen interaction.
- All persons have been using mobile phones for several years, some even considerably longer than ten years.
- Most persons stated to always carry the mobile phone with them when being out of their homes.
- Most of the mobile phones mentioned in the interviews are turned on and used day and night, most persons said that they turn it off only on special occasions, for example while being in concerts or when going to church.

The most obvious finding regarding the last two questions about the usage of additional electronic devices is that most of the test persons stated to own and regularly use a personal computer. Only the oldest two test users do not own and have never used a computer.

In detail, the following answers were given:

- *Do you own a mobile phone?*

  As mentioned above, all of the test persons said to own and regularly use a mobile phone.

- *Which type of mobile phone is it?*

  None of the persons mentioned a smartphone, most of them used a classic phone with keyboard, one person stated to use a special phone aiming at older users.

- *When did you get your first mobile phone, and for which primary reason?*

  The answers to the first question varied between eight and more than 15 years. Not all persons bought their first mobile phone on their own, some got it as a present by relatives, some were given a phone by the company they were working for. Accordingly, some test users stated that the reason they started using mobile phones was that they were asked to by their company. All of them, however, got used to it and decided to buy their own

phone after retiring. Other reasons mentioned several times are the use of a mobile phone as replacement for a landline telephone, and the desire for being available in cases of emergency.

- *What do you typically use the phone for, apart from simple phone calls, and how often?*

  The two oldest test persons said they would use their mobile phone solely for starting and receiving phone calls, one of them instead of a landline phone and therefore very often, the other one rather infrequently. Text messages were also mentioned twice as use case, although one of these two users stated to use the text message service rarely and only for receiving messages. Additional use cases mentioned by the rest of the test persons include the usage as alarm clock, photo camera, calculator, and for paying parking tickets; ordered by the frequency they were mentioned.

- *If you are not in your home, do you usually carry the mobile phone with you?*

  Only one person stated to not regularly carry the mobile phone when out of home.

- *If you are at home, is there a common place where the mobile phone is situated, or do you carry it with you?*

  Answers to this question were diverse. Three persons said the phone was nearly always with them, two said they had a fixed place in their home for putting the phone when not using it at the moment, and the other three stated that they were frequently looking for their phone in different places all around their home.

- *Is the mobile phone always turned on? If not, in which situations and how often do you turn it off?*

  As mentioned above, most users say their phones are turned on all the time. Only one persons said to turn it off by night, one test user stated to be on vacation abroad regularly in which case the phone is turned off. Special events which require the phone to be turned off were mentioned a few times, including concerts and church service.

- *Apart from mobile phones, which other electronic devices do you use?*

  This question showed that all test persons, except for the two oldest, own and regularly use a personal computer. A television set was mentioned similarly often, including additional equipment such as DVD player and hi-fi system.

- *For each of these devices, how often and for which purpose do you use it?*

  Various use cases for personal computers were mentioned, including e-mail, surfing the Internet, writing and printing documents, and displaying and editing photos. All persons owning a PC stated to use it frequently, some even daily.

For interpreting the observations and results of the interaction tests presented in the next section, it should be noted that only one test person stated to use mobile phones only infrequently and to watch TV very often, leading to the conclusion that mobile devices might not be the right platform to use for controlling AAL services for this user. For all the other test users, the

smartphone platform used in the interaction tests seems to be the best solution as AAL user interface device.

3.6.2 Interaction tests

This section presents all observations made during the practical interaction tests. Apart from results concerning the six test cases defined in section 3.4.3, various general observations were made, concerning the general handling of mobile devices using touch screens. Most of them are related to problems occurring to users who are not familiar with the handling of touch screens.

In many use cases that include several buttons on the touch screen being positioned next to each other, users were not able to tap on a particular button without unintentionally hitting a neighboring one. This has two primary reasons:

- Users not familiar with touch screen interaction tend to use their fingernail for tapping on a touch screen instead of the fingertip. Typical touch screens do not react to being contacted by a fingernail, but since the fingertip might unintentionally also contact the screen, only a few millimeters away from the fingernail, the screen might process this as a tap.

- The screen of the test device is rather small, leading to user interface widgets being displayed smaller than on similar devices running the same Android platform. In addition, compared to traditional devices with hardware buttons, simulated buttons on a touch screen lack a physical elevation and a distinct pressure point. Both facts make it difficult to control exactly which button to tap on, if several are positioned in a close neighborhood.

In some cases, the touch screen did not react at all to a user's tap gesture. One reason is, as explained above, the use of fingernail instead of fingertip, another one is pressing too hard. Touch screens expect soft tap gestures by a single fingertip, if the screen is pressed harder than expected the touch is not recognized. One subsequent problem in this case is that users might be confused to not seeing any reaction on the screen, and therefore pressing even harder when trying again.

In contrary, some situations showed the touch screen reacting too fast, also confusing users. Some predefined Android widgets, e.g. the on-screen keyboard, allow long tap gestures, meaning that if one tap on the button creates a certain interaction, if the same button is tapped and held for a longer time, the interaction is executed continuously, until the finger is removed from the screen. It seems that the interval necessary for the screen to interpret a touch as a long tap instead of a simple tap gesture is too short for novice users, resulting in unintentional long tap gestures. One example is if a character button of the on-screen keyboard is pressed for too long, multiple characters of equal type are input in the according text box.

In general, wipe gestures are more difficult for novice users to perform than tap gestures. The nature of a tap gesture is similar to pressing a physical button on a traditional mobile phone, more sophisticated gestures that require motion on the screen need to be learned and trained.

Apart from touch screen interaction, one unexpected observation concerns standard Android user interface widgets: The first screen of the doctor's appointment service allows users to enter search criteria for finding an appropriate doctor, for example by choosing types of insurance

from several check boxes. Initially, all check boxes are selected such that all doctors would be returned by the search engine if a user chooses not to use this criterion. If desired, users may narrow down the search result by deselecting certain check boxes until only one or two types of insurance remain.

In practice, most users thought they would have to tap on the box representing their favored type of insurance, without noticing that as a result all but the desired insurance will be used as search criteria. The check mark displayed next to the caption of a check box was not taken account of. It seems that preselected check boxes, especially when appearing in a group, are not interpreted in the right way by novice users.

Input of a point of time

The test users were asked to input a certain time of day using the six given interaction modes. After each method was successfully handled, they should tap on the confirmation button to load the next test case.

The following observations were made during this test:

- *Input in one text box using the standard on-screen keyboard:*

  The first problem users who are not familiar with Android were confronted with was that it was not clear how to switch between the standard on-screen keyboard and an alternative keyboard showing digits. After explaining which button needs to be pressed to show the alternative keyboard, the input of numbers was not a problem.

  Another problem was that users did not know which character to use as delimiter between hours and minutes. The automatic validation routine was implemented as to accept a wide range of delimiting characters, but test users were unsure whether to try out any potential delimiters or just continue with the next digit.

  The validation routine that displays a small exclamation mark to the right of the input text box and disables the confirmation button if no valid point in time was entered was not recognized by most test users. It seems that they were focused mainly on the on-screen keyboard and on the characters already entered in the text box and did not pay attention to other elements on the screen, such as the confirmation button and the exclamation mark indicating input errors. Reasons may be that the confirmation button is of no use during the input but only afterwards and can therefore be ignored, and that the error icon was apparently not considered as being associated with the text box.

- *Input in two text boxes representing hours and minutes using an on-screen keyboard showing only digits:*

  This test case was considered more practical than the first one by many users, mainly due to the problems of switching keyboards and entering a delimiting character not occurring in this case. As mentioned above, the exclamation mark indicating input errors did not play a role in the user tests since the test persons did not pay attention to it, at least not in a way that affected their interaction with the system.

One comment to this interaction method made by one test person was that after entering a valid value for hours, the focus should automatically switch to the second text box, allowing to seamlessly enter a value for minutes. The necessary interaction for manually switching between the text boxes is indeed not intuitive, two test persons tried to drag the finger from the first text box to the second in order to change focus.

- *Increasing and decreasing values for hours and minutes through + and - buttons:*

  The aspect mainly criticized concerning this interaction method is that the corresponding value is increased or decreased by steps of 1 on each finger tap on one of the buttons. According to most of the test users, the expected and desired behavior would be a continuously increasing/decreasing value as long as the finger stays on one of the buttons. Using the currently implemented version, it is too tedious to tap one button multiple times in order to reach a particular value.

  Another problem was that the + and - buttons were considered too small for some users, thus being hard to tap on.

- *Selection of values for hours and minutes from two simulated wheels:*

  These two test cases showed a general problem of touch screen interaction: As mentioned above, finger wipe gestures seem to be more difficult to perform for novice users than tap gestures. The wipe gestures necessary to scroll up and down the wheels were performed carefully and slowly, wiping in several small steps than in one continuous gesture.

  Additionally, it was not intuitively clear were to start wipe gestures in the best way. The simulated wheels used for these tests should be tapped on with a finger exactly in their center, wiping the finger up or down from this starting point. Due to some reason, some test users started their gestures at the very top or bottom end of the wheel widgets, making it more difficult to perform a full scroll task since focus is lost as soon as the finger leaves the boundary of the wheel widget.

  To one test person it was not clear how to read the currently selected values on the two wheels. The values in the center of the wheels are currently selected, but two neighboring values are shown above and below the current values, totaling in five values being displayed, to emphasize the suggested wheel interaction. One test person thought the first of the five values were the currently selected ones and therefore chose a wrong point in time.

  The difference between the first and second test case using this interaction method, with the second wheel only showing the values 0, 15, 30, and 45 in the second case, was not apparent to some of the users. Most of them stated that the second case might be more useful for the use case of a medication reminder, since the exact minute of an alarm is irrelevant in this case, but only on explicit inquiry.

- *Adjusting clock hands on a simulated analog clock:*

  The main problem of this interaction mode was that the clock hands are too small to allow steady interaction. Since the available screen space of the test device is quite small, also relatively small clock face and hands have to be used. This leads to the action area,

meaning the area around both clock hands in which the user must tap for moving the hands, being too small to tap on. On larger screens, e.g. when using tablet devices, this problem might not occur.

It was also confusing for test users that the clock hands were linked together, meaning that if one hand is moved by the user, the other one also changes its position accordingly. A possible solution to this problem is the segmentation of the interface scenario into two screens, each showing a clock hand with only one clock hand, one for hours and the other one for minutes. This approach might also solve the first problem, since the action area of the clock hand can be larger if only one hand is to be shown.

Another aspect that prevented intuitive user interaction was the fact that it was unclear to many test persons how to switch between morning and afternoon. In general, the direct interaction through a simulated clock face was not approved by all test users.

When asked for the interaction mode they found most useful, most users chose the simple input through on-screen keyboard, but also the selection from wheels and the version with + and - buttons were mentioned twice. All test users definitely disapproved the analog clock interaction.

Input of telephone numbers

The test users were asked to input an arbitrary telephone number using the given interaction modes. After each method was successfully handled, they should tap on the confirmation button to load the next test case.

The following observations were made during this test:

- *Input using either the standard on-screen keyboard or a telephone-like keyboard showing digits:*

  These two input methods caused similar observations as the first two input modes in the first test case. As in the first case, it was difficult for many users to figure out how to switch between the different keyboard types. The second exercise, starting already with a keyboard showing numbers instead of characters, was solved much faster than the first one, not only because the input of numbers was already trained, but mainly due to the exchange of keyboards not being necessary.

  One additional difficulty shown in this test case is caused by the fact that the Android version used by the test device offers three different types of keyboard: the standard one showing characters, a telephone-style keyboard showing digits, and one for entering special characters. A special button on the right side of the keyboard switches between those three. If the user's finger taps this button and stays there too long, this interaction is interpreted by the device as two taps instead of one, resulting in the third keyboard being shown instead of the second one. This behavior confused some of the test persons.

  The same validation routine as in the first test case was used, as soon as the user enters a character that is invalid for telephone numbers a small exclamation mark is shown next to

the text box. In this case, however, this validation routine could never be tested because no invalid characters were entered in the course of the user tests.

One unexpected problem occurred in of the user tests: A test persons said it was unclear which delimiting character to use between area code and phone number. This leads to the assumption that either an effective validation routine is needed that draws the user's attention to any unexpected character entered, or alternatively a routine that is executed after entering the number that removes all imaginable delimiting characters from the entered term, since for further processing the phone number it should only contain digits.

- *Direct interaction with a simulated dial plate:*

  This approach was definitely not approved by the test users. Although imagined as being more intuitive than any traditional input methods for phone numbers, it was declared as not intuitive by some of the test users. The most common problem was that it was unclear where the currently selected digit needs to be dragged in order to be confirmed.

  All test users knew the typical interaction with a dial plate, and most of them mentioned the physical design of a dial plate as its biggest advantage: On the position of every digit, the plate has a small hole in which the user's finger can be positioned. The rotating motion necessary for dialing this particular digit is self-explanatory in this case. On a touch screen, these holes are missing, therefore the expected rotating gesture is difficult and not intuitive to perform.

  Another disadvantage of this method mentioned by some of the test users is the tedious manipulation of the dial plate, since for dialing one single digit the plate has to be rotated first, following by some waiting time until the dial plate has returned to its initial position.

- *Selection from a list of stored contacts:*

  This alternative was approved by most of the test persons, only one user mentioned it was not useful. An additional aspect mentioned by one test user is that such a list of available contacts to choose from is especially useful if provided with photos of each contacts. This could make it even more comfortable and intuitive to select the appropriate contact, e.g. as primary contact person in emergency cases.

  It was also mentioned, however, by several users that this approach is restricted to certain use cases that aim at selecting an existing contact, and not entering a new telephone number.

In summary, the telephone-like keyboard combined with a text box and the selection from a list of contacts were considered the only useful methods by the test persons. Since both methods follow different use cases, it might be useful to follow up both approaches for an implementation.

Selection of date and time

Regarding the first step of narrowing down the period of potential dates on which an appointment shall be reserved, the following observations were made:

- The first method of selecting a single date by choosing the exact day and month from two lists was considered rather inflexible by some of the test users, but nevertheless proved to be the easiest solution to interact with.

- The second method that allows to enter various search criteria for finding an appointment date was extensively discussed with all test persons. Only one user said the first method shown was more useful, the others liked the possibility to provide search criteria. Aspects criticized are that there is too much information presented on the screen and the text size is too small, and that such a degree of accuracy is not necessary, especially for retired persons who are quite flexible usually.

  In addition, the slider widgets used for specifying periods of time were very hard to interact with due to the tap-and-wipe gesture necessary. As an alternative, one user suggested to replace the exact search criteria by choosing whether an appointment shall be before or after noon.

- The calendar-like view brought about two serious interaction problems: First, some users tried to select one of the weekdays in the headline instead of a fixed date, and were confused because those are not selectable. In addition, to switch between months, it was expected by one test user that tapping on the current month's caption would open a list of potential other months, instead of using the buttons to the left/right.

Concerning the following presentation of available appointment dates, the integration of the user's personal schedule was approved by most of the test users, even if some stated that it would only be useful if the phone user actively entered all his personal appointments and that they would not make use of this possibility. Only some details were criticized, for example the fact that it is difficult to distinguish private appointments from the offered ones at first glance, and that it is not clear which appointments are selectable and which not.

Presentation of a list of data items

In the course of the first two user workshops, problems in presenting this test case were observed. The setup phase of the medication reminder example service was presented twice to test users: First iterating through all existing reminders with the possibility of editing them, then displaying a list of existing reminders each of which could be selected an then edited.

The test persons were not aware of the main difference between the two options and the research question connected with this test case. In contrast, users seemed confused due to the execution of two test iterations that seemed equal to them. In addition, it was not possible to focus their attention on the presentation of list items. Instead they concentrated on small details of each list item, for example difficulties in entering a point in time, although these topics had already been discussed in earlier test cases.

The conclusion must be drawn that the setup of this test case was not appropriate to answer the research question it aimed at, namely which type of list item interaction is more suitable. Due to the relatively long period of time this test case took in comparison to the others and to

the confusion it caused to the test persons, it was decided to skip this test from the third user workshop on.

Presentation of location-based data

Concerning the map view showing the nearest doctor's offices in relation to the user's current location, the colored pins were intuitively understood as representing the different locations, and to most users it was also clear without explanation that the pins can be tapped on to select one of the doctor's offices.

The map view itself was criticized as not being intuitive, most users for example did not find buttons for zooming the map. Since an instance of Google Maps[9] was used, the map was easy to operate for users familiar with the Google Maps interface, but hard to use for novice users.

The aspect mainly criticized regarding this test case was that both interaction methods seemed inappropriate for the use case of finding a doctor. Many test persons stated they would not let a mobile phone or an Internet application suggest a doctor they have never been to, but rather follow recommendations from friends or relatives. Therefore both the list of doctors including the exact distance and the map view were considered irrelevant.

Those test users that would use a recommendation service similar to the one presented in the test case were satisfied with the map view, but additional suggestions and wishes were expressed. One test person, for example, suggested to integrate a system for finding the best connections from the current location to a doctor's office using public transportation, and display and compare these schedules instead of a street map. The reason for this suggestion is that the time and effort for getting there is more important than the geographic distance.

Alarms and reminders

The most obvious outcome of this test case was that a continuously playing audio alarm is preferred compared to a short one, and considered the most effective option for alerting the user. All but one test person stated that the short audio signal is not sufficient for alerting the user in a case as important as a medication reminder, and that the best alternative is an audio signal not only playing a bit longer but rather continuously until confirmation. Even on inquiry whether a continuous audio signal might be annoying when it has already been noticed but cannot be confirmed instantly due to being busy with other tasks, the importance of taking some medicine was interpreted more important than any potential annoyance.

The importance of an audio signal as primary alarming method is also emphasized by the fact that several users asked for the audio alert to be played as loudly as possible. The vibration alert was considered not useful especially for older users, since they doubted that the target audience would notice a vibration and react to it. Nevertheless the usefulness of an alternative alert, in addition to the audio alarm, was emphasized by most test persons, especially for users slightly hard of hearing. One person suggested the use of a flashing screen in addition to audio-based alerts.

[9] `https://developers.google.com/maps/`

The preference of visual alerts to fill the whole screen was also expressed clearly by all test users. As the main reason the importance of taking medicine was provided. One of the test persons compared a medication alarm with an incoming phone call: Most common mobile phones interrupt all other applications and use the full screen to indicate an incoming call. Medication reminders should act in the same way, since after confirmation by the user the work in any other applications can be continued anyway. As another reason showing the reminder message using the full screen, the possibility to display the message in characters as large as possible was mentioned by one test person, in order to ease reading the message especially for older users.

Finally, it should be mentioned that the small icon shown in the phone's title bar indicating that the reminder service is running in background was not noticed by the test users. On inquiry, some stated that they knew similar status indicators from other devices, but would not have noticed it with being explained. However, the interaction scenario was clear even without being informed about the background task to all test users.

3.6.3 Conclusions

First, the observations made during the user workshops can be used to define general guidelines for the implementation and the automatic generation of user interfaces for mobile devices aiming at elderly persons, including the following:

- Both content presented on the screen and user interface widgets that require finger interaction should be displayed as large as possible. The former mainly concerns text where a larger font size makes it easier for older people to read the content.

  The latter regards, among others, buttons that can be tapped on, and the aim is to maximize the action area in which a tap gesture is recognized. In addition, such widgets should not be positioned in a neighborhood too close to each other. Both guidelines follow the observation of older users having difficulties with hitting a certain area on the touch screen with their fingers.

- Where possible, those user interface widgets and interaction patterns should be preferred that make use of simple touch screen gestures, with the aim of avoiding both advanced motion gestures and ambiguous gestures. The former includes, for example, finger wipe gestures that are difficult to perform for older users. An example of the latter is the long tap gesture which should be totally avoided whenever it may be confused with the simple tap gesture by novice users.

- User interactions should cause instant feedback to let users know that their input has been accepted and is processed by the device. One example of instant feedback is the clicking sound played immediately after each tap on the touch screen, which can be activated in the Android operating system.

- The observation that users seem to have difficulties interpreting preselected check boxes leads to the suggestion of comparing the generation of a user interface screen with the creation of a paper form: User interface widgets that can be deduced from elements on a

paper form, such as text fields and check boxes, should be used in the same way as they are expected to occur in a paper form. This includes avoiding text boxes already filled in and preselected check boxes.

In addition, an important aim of the user workshops was to answer the three research questions defined in section 3.3.2:

For each of the user interaction scenarios identified above, which of the suggested solutions is most suitable for the target group of the system?

This question has to be answered individually for each interaction scenario, following the test users' suggestions of which interaction modes they considered most useful. In addition, the user interface guidelines defined above should be taken into account.

For example, for the *input of a point in time* scenario, the optimum solution might include two text boxes, one for hours and one for minutes, in which the desired values can either be entered manually using an on-screen keyboard showing digits, or alternatively changed by additional + and - buttons. If appropriate for the use case, as in the medication reminder service, this buttons would increase or decrease the values in certain larger intervals rather than just adding or subtracting 1.

For entering telephone numbers, a simple text box and an on-screen keyboard showing digits should be used. If appropriate for certain use cases, an option for alternatively select the telephone number of an existing contact can be provided.

When selecting appointment dates from a given list, they should be displayed in relation to the user's personal schedule, if available. Since the calendar-like view was not approved by all test users, a simple list containing all potential appointment dates should be preferred. For each list item, it can be marked if the respective appointment conflicts with one of the user's personal appointments.

Since no consistent opinion for the treatment of location-based data could be found, the easiest imaginable solution should be used as standard, usually meaning the display of all information available as text. Depending on the user interface device, alternative options may be provided that must be explicitly activated by the user, for example the possibility to show the location of a street address that is displayed as text also in a map view on devices with relatively large display.

For alerts and alarms, a continuously playing sound signal should be used, accompanied by a vibration alert and a visual signal such as a flashing screen. The message to be conveyed should be displayed as large as possible, using the full screen space.

How much information should be shared between back-end service and user interface device?

Whenever possible, the back-end service should offer the user interface device additional information beyond the simple data type when presenting information to or requesting data from the user. However, it must be specified which type of information the user interface device can expect to get. For the input of data, this can include information about which input is valid for a certain data item, for example a regular expression encoding valid input for text data, or minimum and maximum values for numeric data types.

In addition, semantic data specifying the purpose of a certain data item can be included. This might help for outputting data, since the user interface device can present and display the information in the way that is suited best, as well as for inputting data, as the user interface device can then provide the user with the input method that fits the type of the information best. To ensure compatibility between back-end service and user interface device, potential values should be predefined, for example by specifying a set of content types which are more detailed than simple data types.

One example could be the content type *telephone number* which is of data type `String` but can be treated in other ways than normal text. For example, when outputting text that is known to represent a telephone number, a mobile phone used as user interface device could provide a direct hyperlink that allows the user to dial this number.

For inputting data of type `String`, a text box and an on-screen keyboard will be used typically by graphical user interface devices. However, if the data to be entered is known to be a phone number, a special on-screen keyboard can be used that only shows digits, and the option to select an existing phone number from the user's list of contacts can be provided as alternative.

Are potential users willing to allow services to access their private data?

Concerning the two use cases of selecting a contact's telephone number and comparing potential appointment dates with the personal schedule, users did definitely approve the integration of their personal data.

An important aspect to take into account whenever building upon data that is not provided by the back-end service but rather by the user interface device is that such data may not in all cases be available. Therefore an alternative user interaction solution must be provided in any case. Examples include users that do not use their phone's contacts application or calendar, in which case it must nevertheless be possible to enter a phone number or select an appointment date.

CHAPTER 4

Prototype

4.1 Concept

As part of this thesis, the prototype of a user interface generation system was developed. Most of the existing User Interaction Description Languages presented in section 2.3 enable the description of services and abstract user interfaces that can be transformed into concrete production user interfaces in a hierarchical approach. Some of the languages focus on the description of services and the interaction those services offer, excluding strategies to actually generate user interfaces that make use of those service interactions.

The system presented in this chapter combines the two approaches mentioned above. It focuses on the description of user interactions rather than user interfaces, but nevertheless allows the automatic creation of user interfaces for accessing those interactions. It contains a framework for defining remote services, and applications for different user interface devices that implement those services. The prototype implemented as part of this thesis includes one exemplary application that allows the generation of user interfaces for mobile devices running the Google Android[1] operating system.

This chapter describes the interaction concept used as basis for the prototypical system. Abstract architecture and actual implementation of both the framework and the exemplary user interface application are presented in detail. Finally, potential extensions and adaptations to the system are discussed that might be incorporated into future versions of the user interface generation system.

4.1.1 Interaction-based user interface generation

One of the requirements of a User Interface Description Language is that it must be able to describe user interfaces independent of modalities, platforms or toolkits. This requirement is necessary to ensure that for interacting with the system, the user can choose from a maximum number of devices. Different devices might use different platforms, and different platforms might use different modalities of interaction (e.g., visual interaction, tactile interaction, voice-based interaction, etc.). The requirement of support of a maximum number of interaction modalities implies that a UIDL must define user interfaces in a sufficiently abstract way in order to not prohibit the generation of a concrete UI implementation on a specific device, using a specific modality.

This leads to the idea of rather defining interactions instead of concrete interfaces. Each task a user interface offers the user to accomplish should be decomposed in smaller parts, each part describing a simple interaction between user and system. These interactions should be defined as generic as possible in order to allow re-use of interactions in several tasks.

[1] `http://www.android.com`

4.1.2 Basic assumptions and definitions

For implementing an actual software solution following this approach, some basic terms were defined and assumptions about user interaction procedures were made, in order to formalize the interaction between the user (through a user interface device) and a remote service. This section lists and discusses this assumptions and definitions.

1. An AAL *service* is defined as a piece of software that allows the user to accomplish a certain task inside an AAL environment.

2. Each service is defined as a sequence of *interactions*. These interactions may occur between the service and any end device, or between the service and a user.

   The former includes interactions with all end nodes apart from user interface devices, including for example sensors, storage devices, database systems, web servers, etc. Since the focus of this thesis in on user interactions, the assumptions made in this section only apply to interactions between services and users.

3. Each interaction between human (in this case, the user of a service) and computer (in this case, the service) can be described as transmission of data from one partner (user or service) to the other. From the point of view of a service, this means either

   - the *output* of data to be received by the user, or
   - the request of data *input* from the user.

4. In practice, every interaction between user and service will be realized through a *user interface device*. In case of an output interaction, this device is responsible for presenting some kind of data to the user, while in case of an input interaction, it converts user actions to data the service is able to process.

   For each service to be executed, the user decides which user interface device he or she wants to use for interacting with this specific service, by simply starting the service from the respective device.

5. Since a service is described as a sequence of multiple interactions, it is necessary to define *relations* between interactions. For the framework presented in this chapter, it is sufficient to define a distinct order in which the interactions take place, or alternatively define multiple interactions as simultaneous meaning that they might be carried out at the same time if possible. Section 4.1.3 explains the permitted relations in detail.

   More complex types of relations need not necessarily be defined on the user interface level. One example is the definition of distinct periods of time between two interactions in addition to their sequence, which is not needed in this case since timing issues can be covered completely by the services.

6. Each interaction can be defined more precisely by the type of data that is input or output in the course of the interaction. This is defined as one out of a predefined set of *data types*.

Basic data types defined in this set could for example include boolean data, numeric data, text, telephone number, geographic data, etc.

4.1.3 Information flow

As soon as a service has been started by the user, its application flow is executed. As mentioned above, a service might communicate with sensors, storage devices, databases, web servers, user interface devices, etc. The framework presented in this chapter does not make any assumptions about a service's communication with end points other than user interface devices, nor does it restrict internal operations that might be carried out by a service. However, when it comes to a service's communication with the currently active user interface device, a strict procedure is to be followed.

Between other operations, at certain points in the application flow it will be necessary for a service to present some information to the user, or request the user to input certain data. This is accomplished through output and input interactions. The service initiates an output interaction with the user interface device in order to ask it to present some data to the user. Likewise, the service initiates an input interaction with the user interface device to ask it to request certain data from the user. The overall control over the application flow is at the service side, the user interface device is responsible only for presenting information and requesting data in the best way.

This approach follows the concept of a central service controlling the application flow, supported by highly specialized nodes responsible for certain strictly delimited functionalities. This brings about the following advantages:

- A single central control unit reduces complexity and improves speed and reliability compared to a system of multiple nodes all of which are granted control over the whole system.
- Outsourcing of specific functions ensures that those functions are processed using the highest possible quality level. For example, a user interface device is able to decide how to present data of a certain type to the user better than a remote service. This decision can be based on several parameters which are not available to the service but only to the user interface device, including the technical abilities of the user interface device and the user's preferences to work with this exact user interface device.

  Usually, a device that is well known and commonly used by the user is the first choice for an AAL user interface device. In this case, it can be assumed that the device has already been customized to fulfill the user's preferences, implying that those preferences need not even to be set explicitly to the AAL environment - if they are known by the user interface device, any AAL service can implicitly make use of them.

Commands and interactions

A service may send commands to the user interface device in order to inform it about an interaction to be carried out. The following list summarizes the four possible intentions a service

might express by sending a command to the UI device:

- The service might ask the UI device to output data to the user through the user interface, meaning to initiate an output interaction, or
- the service might ask the UI device to request an input of data from the user through the user interface, meaning to initiate an input interaction, or
- the service might inform the UI device that an interaction has timed out and has therefore been canceled, or
- the service might inform the UI device that the task the service should accomplish is finished and that no more commands will follow from that service.

In case of an output or input interaction, the command the UI device receives from the service contains the data type to be transmitted to or requested from the user, and (in case of an output interaction) the data itself. In addition, it may contain supplemental information about the data. This includes both formal restrictions, for example the allowed minimum and maximum size of the data item to be entered by the user, and semantic information describing the nature of the expected data item.

For each command, a concrete widget is generated based on the toolkit (implementation language) that is used by the UI device. For requesting input of boolean data on a mobile device using graphical modality, for example, a group of radio buttons defined in XAML, Java Swing, HTML, etc. might be used as widget. For using voice-based modality for an output operation of a text data type, a text-to-speech generator might be defined as widget. The decision how exactly data is presented to or requested from the user is completely made by the UI device, based on the data type, the user's preferences and, if defined, the semantic information which describes the data item.

In case of an output interaction, the UI device is also responsible for filling the widget with the actual data. In case of an input interaction, the UI device is responsible for validating the input data acquired from the user to check if it fits the requested data type and fulfills the formal restrictins, if defined. Then it transforms the data to a format readable by the service (meaning, to the exact data type that was requested by the service).

If the requested interaction was of type output, the UI device notifies the service when the data was completely transmitted to the user interface (using voice-based output for longer texts, for example, might take some time, in which case the service should wait until this operation is finished before continuing its operation). Then the service continues by sending the next command, which might be an input interaction requesting feedback from the user, or another output interaction. Alternatively, the service can continue by accomplishing interactions with other services or devices, or just by waiting a certain period of time.

If the requested interaction was of type input, the service waits until the data acquired from the user is transmitted by the UI device, and then continues its operation by processing the input data. Since it might take a long period of time for the user to input the requested data, the service is able to process interactions with other services or devices in the meantime. In addition, the service might define a timeout period, after which it no longer waits for the requested user input,

but continues by sending further commands. The next command in that case could be an output interaction, reminding the user that feedback is required, followed by a repeat of the original input operation.

Figures 4.1 and 4.2 schematically show the communication between service and UI device in case of an output and input interaction, respectively.

Figure 4.1: Communication between service and UI device for initiating an output interaction

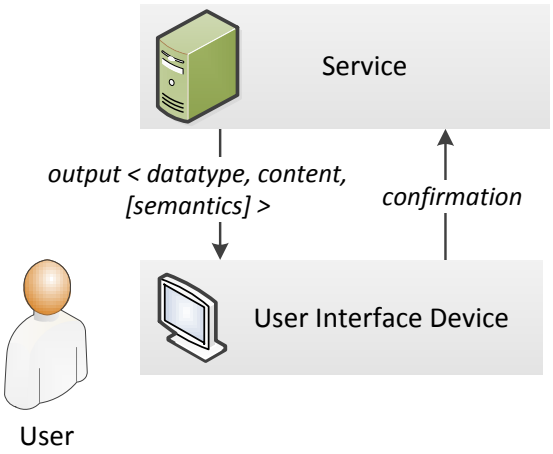
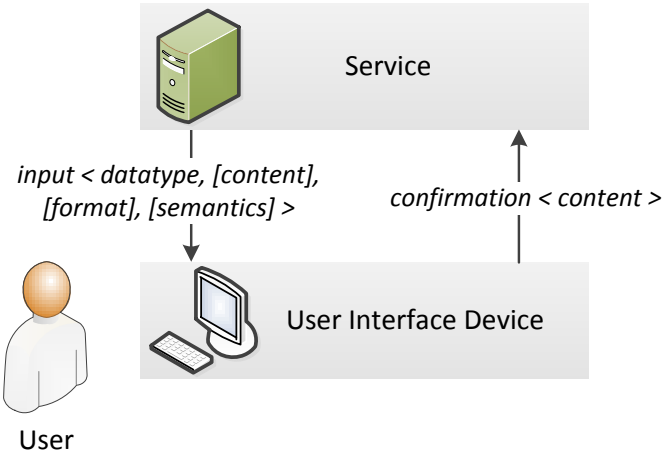


Figure 4.2: Communication between service and UI device for initiating an input interaction



It should be noted that the UI device is only responsible for presenting data to the user and requesting data from the user. Further operations, for example timing issues, are completely in the responsibility of a service. This restriction might lead to undefined states in the communication between service and user interface device, therefore the third type of command is incorporated: the *cancel* command:

A service may internally define a timeout period after which an input interaction is canceled and the service no longer waits for user input. In this special case, the input request is still active on the user interface device while already being canceled on the service layer. If the user inputs data after the cancellation, it will still be forwarded to the service if the UI device is not aware of the cancellation, thus leading to an undefined state in the service due to receiving unexpected data. To avoid such a case, the service must always send a *cancel* command immediately after canceling an interaction due to time out. This is necessary to ensure that the user interface is informed about the cancellation and can react by also canceling the ongoing process. In case of a graphical user interface, for example, an appropriate reaction would be to remove all information from the screen and refuse to accept input from the user until the next input interaction is started by the service.

Relations between interactions

Since interactions are carried out one after the other to perform a service functionality, every interaction (except for the first and the last one of a service) has exactly one predecessor and one successor in time, thus implicitly defining a sequence in which they are carried out. Alternatively, multiple interactions may be defined as simultaneous. This relation is only allowed for interactions that do not depend on each other, because it means that the order in which these interactions are carried out does not matter and may be defined by the user interface device at runtime.

The advantage of allowing simultaneous interactions is the ability to generate advanced user interfaces that present several output messages (or request several input variables) at the same time. Graphical user interfaces, for example, can take advantage of the *simultanous* relation by presenting a dialog window to the user that contains several widgets, each representing one interaction. Similarly, voice-based input/output systems may carry out multiple input interactions at the same time by asking the user to input several data sets, if those data sets can be clearly distinguished and uniquely assigned to the according interactions by the system. This requirement of distinguishable data sets is fulfilled, for example, if all of the simultaneous interactions request different data types.

In practice, defining a group of interactions as simultaneous is useful if they accomplish semantically similar tasks. For example, when requesting day, month, and year of birth from a user, those three input interactions should be defined as simultaneous. A graphical user interface would then present tree input widgets on the same screen, thus appearing more intuitive to the user than three separate screens for each data.

It is important to note that the permission of simultaneous interactions does not restrict the number of supported modalities or user interface devices in any way. Platforms that don't support the concurrent execution of interactions can still execute them one after the other, not being bound to a specific sequence.

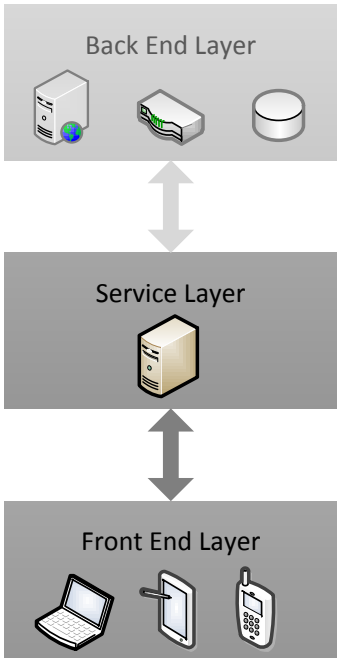
4.2 Architecture

This section describes in detail the different software components the prototype of the user interface generation system comprises. In addition, a general model of the different hardware devices on which these components are executed in introduced.

4.2.1 Hardware structure

From an architectural point of view, user interface devices and services as described above reside on two different hardware layers, as illustrated in figure 4.3.

Figure 4.3: Hardware architecture: the communication between service layer and back end layer is not controlled by the user interface generation system presented in this thesis



The *service layer* is responsible for running AAL services and controlling their application flow. It may communicate with user interface devices and other hardware devices, outsource certain functionalities to these devices and process information delivered by these devices. Usually, the service layer contains only one computer hosting all services.

All user interface devices reside on the *front end layer*. They receive commands from the services running on the service layer, react to these commands for example by initiating output and input interactions, and report the results back to the service layer. As illustrated in figure 4.3, different types of devices may be used as UI devices, including for example mobile devices, TV sets, personal computers, etc. From an architectural point of view, all these are situated on

the front end layer.

When starting a service, the user must decide which of the available UI devices to use for the current execution of this service. In the course of the service execution, all commands from the respective service are forwarded to and processed by this exact UI device.

In addition, services might make use of supplemental hardware devices that provide special functionalities, such as sensors, storage devices, database systems, web servers, etc. All these additional hardware nodes are abstracted by the *back end layer*. However, since the user interface generation system presented in this thesis does not cover the integration of such back end devices, the back end layer and all communication between service layer and back end layer is ignored in the rest of this chapter.

4.2.2 Software components

The user interface generation system developed as part of this thesis consists of four parts, namely four software components that communicate with each other, as illustrated in figure 4.4:

- The *server application* hosts a registry of available services. It may receive commands from the user interface application in order to start one of these services. The server application resides on the service hardware layer.

  The implementation developed as part of this thesis includes a primitive server application that allows basic testing of the system. It contains a list of two example services, one of which is started by instruction of user interface applications.

- The *user interface application* is responsible for presenting data to, and requesting data from the user. It requests the server application to start a certain service and then communicates with this service. It received commands from the service, telling it to initiate or cancel interactions, as described in section 4.1.3.

  Since the user interface application transforms abstract data types received as part of output or input interactions to actual toolkit widgets, different user interface applications must be developed for each user interface device and each operating system executed on these UI devices. Therefore an AAL environment implemented using the system presented in this thesis will usually contain multiple instances of the user interface application, one for each UI device present on the front end hardware layer.

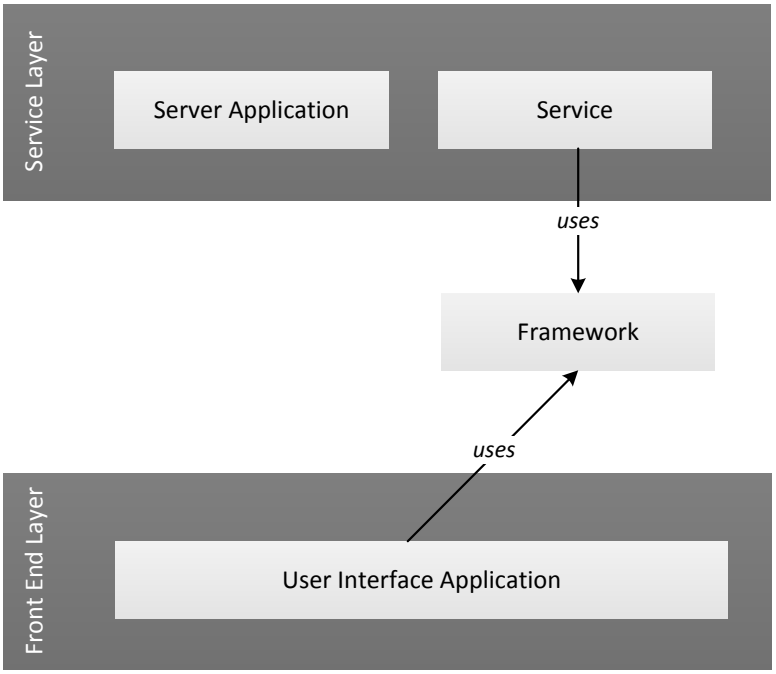
  The implementation developed as part of this thesis includes one example implementation of user interface application, developed for mobile devices - smartphones and tablet devices - running the Android operating system. This implementation is described in section 4.4.

- The *services* allow users to accomplish certain tasks inside an AAL environment. To fulfill these tasks, they communicate with users through user interface applications running on UI devices. Services reside on the service hardware layer.

  The implementation developed as part of this thesis includes two example services. Their implementation is discussed in detail in section 5.2.

- The *framework* is a software component that is instantiated and used by both services and user interface applications. It provides certain functionalities that are useful for these two components, thus reducing complexity and allowing seamless collaboration by providing standardized procedures. The framework can therefore not be assigned to a specific hardware layer, instead it acts as assistance for several components residing on different hardware layers.

Figure 4.4: Software components, assigned to the respective hardware layers



This section explains the collaboration of those software components when used together in an AAL environment:

The server application is continuously running and waits for a user interface application requesting a connection. Vice versa, when the user starts the user interface application on a specific UI device, it requests a connection with the server application. As soon as this connection has been established, the communication between service hardware layer and front end hardware layer mainly consists of two parts which can be decomposed into five basic steps:

1. First, user interface application and server application communicate with each other:

   a) The user interface application requests a list of all available services from the server application.

   b) The server application delivers this list of services.

   c) The user interface application asks the server application to start one of the listed services.

2. As soon as a service has been started, this service communicates with the user interface application:

   a) The service sends a command to the user interface application in order to initiate or cancel an output or input interaction, as specified in detail in section 4.1.3.

   b) When the command received by the user interface application was processed, a confirmation is sent back to the service.

   Steps 2a and 2b are repeated until the service sends a command declaring that the service task is finished and no more commands will follow from this service.

4.2.3 The framework's role

As mentioned above, the framework is not implemented as a stand-alone software component, it rather acts as assistance to the service and user interface application components. As such, it fulfills four purposes:

- It defines constants that should be used by services and user interface applications and therefore guarantees standardization in the communication between those two components.
- It defines a set of data types that should be used by services and user interface applications. This ensures also maximum standardization in the collaboration of those two components, as well as exchangeability of user interface applications: All user interface applications, no matter for which UI device they were developed, must act exactly the same to the services and use the same data types in order to ensure that the same AAL task can be accomplished using any UI device.
- It defines a common pattern for implementing services, in order to guarantee exchangeability of services: All services must use the same predefined communication channels to ensure that any user interface application can collaborate with any service.
- It provides standardized procedures for creating and sending commands and confirmations, and for initiating and canceling interactions.

The actual implementation of the framework developed as part of this thesis is discussed in detail in the next section.

4.3 The framework

Similar to the example AAL services and the user interface application for Android devices developed as part of this thesis, the framework prototype was also implemented using the Java programming language. This section explains in detail how the four core functionalities of the framework component listed in section 4.2.3 are realized in the actual implementation.

4.3.1 Constants

One purpose of the framework is the definition of standard constants to be used by the other software components taking part in an AAL environment. The prototype of the framework implemented as part of this thesis specifies two classes of constants: *Presentation Characteristics* and *Device Characteristics*.

The former includes flags defining the detailed presentation of output and input interactions by UI devices. This class contains only two constant values: `PRESENTATION_SINGLE` and `PRESENTATION_GROUP`. For each output or input interaction handled by the UI device, only one of these two flags may be set. This indicates whether a specific interaction is presented on its own, or as part of a group of interactions defined as simultaneous. In the second case, the UI device might present multiple of these interactions at the same time, if possible.

The *Device Characteristics* class specifies flags indicating which special features a specific UI device provides. In the prototype version, this includes four constant values, several of which may be combined to specify a UI device in detail:

- `ABILITY_PHONECALL` indicates that a UI device is able to make and receive phone calls. For example, when developing a user interface application for mobile devices, this flag can be used to distinguish smartphones from tablet devices, since smartphones are able to make phone calls by definition, while most tablet devices are not.

- `ABILITY_SENDMAIL` indicates that a UI device is able to compose and send e-mail messages. This depends on several features, for example the presence of an e-mail client application on the device and the availability of a hardware module enabling network connections.

- `FEATURE_CALENDAR` indicates that a UI device features a calendar application.

- `FEATURE_CONTACTLIST` indicates that a UI device features an application allowing the user to store contact details.

Although the set of predefined constant values is rather small in the prototype version of the framework, it allows the addition of supplemental values by custom implementations of user interface applications.

4.3.2 Data types

In accordance to the guidelines listed in section 3.6.3, the data types were defined and implemented in order to encode as much information about the underlying data as possible. This is achieved through two different approaches:

First, the selection data types defined by the framework follows a typical user's understanding of which different types of information exist, rather than pure technical characteristics such as memory consumption or processing capabilities. Therefore the set of predefined data types differs from that of common programming languages.

For example, there is only one data type holding numeric data, no matter how large the number encoded by the data type is and whether is is a decimal number or not, because a distinction based on those technical details - size and number of decimal places - is not transparent to the user. Instead, several special data types encoding typical data used in computer services are defined, for example types for specifying telephone numbers and e-mail addresses, although all these types of information might internally be encoded by using `String` data types. This ensures that user interface applications are able to present information in the best way, namely in that way that is most suitable to the user.

Second, for each data type additional information may be defined. This includes both formal restrictions, such as the minimum and maximum size of a data item, as well as semantic information. The latter's purpose is to define the nature of a data item in detail, for example by further subdividing a given data type into several classes.

From a technical point of view, each data type is based on a built-in Java data type that holds the content of the data item to be encoded. To ensure that conversions between this underlying base type and the framework-specific data types are possible, all predefined data type implementations are derived from the common Java interface `Type` which constrains them to implement to the basic methods `get` and `set`.

The `set` method allows user interface applications to convert data entered by the user to the requested data type. In the course of this conversion, the data entered must be checked against the formal restrictions, if defined. If invalid data, meaning data that does not fulfill the formal restrictions, is to be converted, an exception of type `IncompatibleTypeException` is thrown which is specified by the framework. Vice versa, the `get` method allows services to convert framework-specific data types to basic Java data types in order to further process the information that was provided by the user.

In addition, most data types define multiple constructor methods that conveniently allow to define data types and specify content, formal restrictions, and semantic information at once.

The prototype version of the framework defines the following data types:

- The data type `Text` is used for all data for which none of the other, more specialized data types such as `PhoneNumber` and `EmailAddress` is appropriate. Its base type is the Java type `java.lang.String`. As formal restriction, minimum and maximum number of characters may be defined. In addition, a term encoding a regular expression pattern might be defined, which must match the content of the `Text` data type.

If an instance of a data item of type `Text` is created without specifying the content, the empty String containing 0 characters is used as default. This default value can also be accessed directly though the public constant `DEFAULT_VALUE` defined by the implementation of the `Text` data type.

- The data type `Boolean` represents boolean data and is based on the Java type `java.lang.Boolean`. Its default value, which can also be accessed directly through the public constant `DEFAULT_VALUE`, is `false`. No formal restrictions or semantic information may be defined.

- The data type `Number` represents any numeric data. It is based on the Java class `java.math.BigDecimal` in order to cover all potential numeric values. The data type provides a variety of similar constructor methods, allowing the direct instantiation of a `Number` data item based on variables that are of type `int`, `float`, `double`, or `java.lang.String`.

  As formal restrictions, the minimum and maximum numeric value, the maximum number of decimal places, and the desired accuracy may be defined. For example, for requesting the input of a certain amount of minutes in one hour from the user in steps of quarter to an hour (0, 15, 30, 45), a `Number` data type can be used with minimum and maximum values of 0 and 59, respectively. The maximum number of decimal values would be set to 0, and the accuracy value to 15.

  In addition to the constant `DEFAULT_VALUE` which is 0 in this case, the data type `Number` defines two more constants: `MIN_VALUE` and `MAX_VALUE`. Those represent the smallest and largest numeric values the Java programming language can process using the `double` data type, respectively.

- The data type `Time` is used for representing either a point in time or a period of time that is shorter than 24 hours. It is internally based on the Java type `java.lang.Integer` and stored as the number of milliseconds that represent the point of time or period of time to express.

  As formal restriction, the minimum and maximum value and the accuracy may be defined. The accuracy value specifies whether the smallest unit to be used for expressing the desired point in time or period of time should be milliseconds, seconds, minutes, or hours.

  Also this data type provides the constant `DEFAULT_VALUE` which equals the point in time when the current instance of the data item of type `Time` has been instantiated. In addition, `MIN_VALUE` and `MAX_VALUE` constants are provided. `MIN_VALUE` is 0, while `MAX_VALUE` equals the number of milliseconds that represent 24 hours.

- The data type `Date` is similar to the type `Time`. It allows the definition of a date as day, month and year. Like the type `Time`, it allows to restrict values by setting maximum and minimum values and the accuracy. Also this data type provides the constants `DEFAULT_VALUE`, which equals the point in time when the current instance of the data item of type `Date` has been instantiated, as well as `MIN_VALUE` and `MAX_VALUE`.

- The data type `GPS` is used to encode a geographic position using GPS (Global Positioning System) coordinates. It is based on an array of two values of type `java.lang.Double` that represent latitude and longitude coordinates. Four values - minimum latitude, minimum longitude, maximum latitude, and maximum longitude - may be specified for defining an area inside which the geographic position must be located.

  In addition, this data type allows the definition of semantic information. The flag `IS_CURRENT_POSITION` may be set to indicate that, in case of an input interaction, the user's current position is requested. In that case, the user interface application can directly return the current position (if the UI device contains a location retrieval sensor) without even asking the user for data input.

- The data type `SingleChoice` contains a list of items, one of which may be chosen by the user. Internally, it is based on the Java type `java.lang.Integer` and an array of `java.lang.String` objects. The items in this array may be set at instantiation and represent the list items. The `java.lang.Integer` variable contains the index of the list item that has been chosen by the user or preselected by the service, or -1 in case that none of the list items is selected.

- Similarly, the data type `MultipleChoice` contains a list of items, of which several may be chosen by the user. Internally, it is based on a Java list of `java.lang.Integer` values and an array of `java.lang.String` objects. The items in this array may be set at instantiation and represent the list items. The `java.lang.Integer` variables represent the indices of the list items that have been chosen by the user or preselected by the service. As formal restrictions, the minimum and maximum number of list items that may be selected by the user can be specified.

- The data type `PhoneNumber` is a specialized data type that is used for storing telephone numbers. It is based on the Java type `java.lang.String` and similar to the framework-specific data type `Text`. As formal restriction, minimum and maximum number of characters may be defined. In addition, an area code may be specified. If a data item of type `PhoneNumber` is used in an input interaction, this area code is displayed together with the phone number to be entered by the user. The user is therefore restricted to enter only phone numbers belonging to this exact area, and no misunderstanding can occur whether the phone number to be entered must include an area code or not.

  When using the `set` method to set the content of a data item of type `PhoneNumber`, it is automatically checked if the new value has the typical format of a phone number (meaning, it contains only digits, blank spaces, and certain delimiting characters). If not, an exception of type `IncompatibleTypeException` is thrown. In addition, this data type provides a second getter method `getInternal` which returns the phone number in a format from which all blank spaces and delimiting characters have been removed. This can be used by services and user interface applications, for example, for directly calling the given phone number.

  Two additional flags may be set as semantic information: In case of an input interaction, the flag `IS_USERS_NUMBER` indicates that the phone number requested is the user's

personal phone number. If the UI device currently used is able to send and receive phone calls, as for example smartphones and some tablet devices do, the user interface application might just return the UI device's phone number without even asking the user for input.

The flag `IS_KNOWN_BY_USER` indicates that a telephone number of a person that is known by the user is requested for input. In that case, the user interface application might offer the possibility to select a telephone number from the user's contacts list instead of entering one manually.

- Similarly, the data type `EmailAddress` is a specialized data type that is used for storing e-mail addresses. It is based on the Java type `java.lang.String` and similar to the framework-specific data type `Text`. As formal restriction, minimum and maximum number of characters may be defined. In addition, the domain name may be specified. If a data item of type `EmailAddress` is used in an input interaction, this domain name is used and the user needs only to enter the first part of the e-mail address.

  When using the `set` method to set the content of a data item of type `EmailAddress`, it is automatically checked if the new value has the typical format of an e-mail address. If not, an exception of type `IncompatibleTypeException` is thrown. In addition, also this data type provides the `getInternal` method.

  The semantic flag `IS_KNOWN_BY_USER` indicates that an e-mail address of a person that is known by the user is requested for input. In that case, the user interface application might offer the possibility to select an address from the user's contacts list instead of entering one manually.

- The data type `Command` may only be used in input interactions. It is based on the Java type `java.lang.Boolean` and therefore can only contain the values `true` and `false`. Is is used whenever the user is expected to trigger a command rather than enter data or selecting information. When using a UI device that features a graphical user interface, for example, this would be done by clicking a command button. For evaluation, the service needs to interpret the data type's boolean content: It is `true` if the user chose to trigger the command, and `false` otherwise.

4.3.3 Commands & confirmations

In practice, the data types defined above are used in combination with output and input interactions. To initiate such an interaction, a service must send a command to the user interface application. In the framework implementation, those commands are represented as Java objects called *operations* that derive from the abstract class `Operation`.

The framework provides the following classes that are derived from `Operation`:

- `OutputOperation` represents a command with the intention of initiating an output interaction. It must contain a unique identifier and a data item. In addition, a caption may be defined. The identifier is needed because the user interface application sends back

a confirmation to the service. In this case, the identifier is used to assign confirmations received to operations sent.

The data item must be an instance of one of the predefined data types. Through this, data type, content, formal restrictions and semantic information may be defined at once since they are all specified by an instance of any data type. The optional caption value should contain a text describing the content of the data to be output to the user, and is presented by the UI device in combination with the actual data.

- `AlertOutputOperation` is of the same structure and has the same purpose as `OutputOperation`, but in addition it indicates that the data to be output is of importance and should be presented to the user instantly. User interface applications might react to such an operation by playing alarm sounds and interrupting other currently running services in order to alert the user.

- `InputOperation` represents a command with the intention of initiating an input interaction. Like an `OutputOperation`, it must contain a unique identifier and a data item, and it may contain a caption. Again, the identifier is used to assign confirmations to operations. The data item must be an instance of one of the predefined data types. It may contain a content, formal restrictions and semantic information. The optional caption value should contain a text describing the content of the data to be requested from the user, and is presented by the UI device in combination with the actual request.

- `AlertInputOperation` is of the same structure and has the same purpose as `InputOperation`, but in addition it indicates that the data should be requested from the user as soon as possible. User interface applications might react to such an operation by playing alarm sounds and interrupting other currently running services in order to alert the user.

- `GroupOperation` represents a command indicating that several simultaneous interactions shall be initiated. As specified in section 4.1.3, simultaneous interactions may be presented by the user interface application at once or in any arbitrary order, therefore they must be sent to the user interface application at once.

  Also a `GroupOperation` contains a unique identifier that is used to assign confirmations to operations, and it may contain an optional caption that describes the group of interactions to be initiated. Apart from that, it holds a list of objects of type `OutputOperation` or `InputOperation`. This implies that those two types may be nested: Each operation contained inside a `GroupOperation` specifies its own identifier and, if desired, caption. To ensure that only objects of type `OutputInteraction` or `InputInteraction` are used as content of a `GroupOperation`, the Java interface `Presentable` is introduced that is implemented only by those two types of operations.

- `AlertGroupOperation` is of the same structure and has the same purpose as `GroupOperation`, but in addition it indicates that the current group of output or input interactions is of importance and should be presented to the user instantly. User interface

applications might react to such an operation by playing alarm sounds and interrupting other currently running services in order to alert the user.

- `CancelOperation` represents a *cancel* command and is used to inform the user interface application that an interaction which was initiated by a previously sent operation shall be canceled. This operation contains only one member value which represents the unique identifier of the operation to be canceled.

- `EndOperation` does not define any member objects at all. It is sent to the user interface application to indicate that no other operations from the current service will follow and the user interface application can therefore be closed.

All operations, except for `CancelOperation` and `EndOperation`, require the user interface application to inform the service that the operation has been received and processed. This is done by sending a confirmation from the user interface application back to the service. In the framework implementation, those confirmations are represented as Java objects called *results* that derive from the abstract class `Result`. All those results contain a member value representing the unique identifier of the operation they are related to. This way, the service can easily assign results received to operations sent.

The following three types of classes derived from `Result` are defined by the framework:

- `OutputResult` is used as confirmation for operations of type `OutputOperation` or `AlertOutputOperation`. It does not declare any member objects apart from the identifier of its related operation, since its only purpose it to inform the service that the operation has been processed and the data has been presented to the user, without including additional data.

  A result of type `OutputResult` is sent to the service as soon as the information contained in the original output interaction has been fully presented to the user. In case of a UI device using graphical user interfaces, this will be the case instantly after displaying the information on the graphical screen, while when using voice-based output modality that uses a text-to-speech converter, the user interface application waits until the information has been read out to the user.

- `InputResult` is used as confirmation for operations of type `OutputOperation` or `AlertOutputOperation`. In addition to the identifier of its related operation, it must contain a data item that is of the same data type as the operation it acts as confirmation for. This data item contains the data that was entered by the user.

  A result of type `InputResult` is sent to the service as soon as the user has finished entering the requested information, and this information has been validated to fulfill the formal restrictions.

- `GroupResult` is used as confirmation for operations of type `GroupOperation` or `AlertGroupOperation`. In addition to the identifier of its related operation, it contains a set of data items - one for each nested operation the `GroupOperation` it belongs

to contains - accompanied by their unique identifiers. This allows the service to identify each of the nested operation's results.

A result of type `GroupResult` is sent to the service not until all of the output and input interactions initiated by the original operation have been fully processed.

4.3.4 Services & connections

In addition to data types, operations and results, the framework provides a common pattern for the easy implementation of services, and a helper class for enabling communication between services and user interface applications.

The latter is implemented by the class `Connection` which is realized as a singleton class meaning it can be instantiated only once. Usually, each service will request access to the single `Connection` instance at startup, open a connection to the currently active UI device, use this connection for sending operations and receiving results over the full application life cycle, and finally close the connection.

For opening a connection to a UI device, the `Connection` class provides the `connect` method that expects the port identifier to be used for the connection being passed as parameter. For closing the connection, the `close` method is used. Since the prototype implementation is developed as part of this thesis with the primary purpose of being a proof of concept rather than a solution for productive environments, the focus is on user interactions rather than on networking concepts. Therefore, the framework uses simple socket connections for the communication between services and user interface applications, and it does not guarantee secure connections and full exception handling.

The `Connection` class provides the `send` method to send operations to the currently active user interface application. When a result is received from the user interface application, the `receive` method is called automatically. This method calls the respective callback method of the service implementation, depending on which type of result was received. The `receive` method also checks the identifier of incoming results. If it does not match with the identifier of the previously sent operation, an exception of type `UnexpectedResultException` is thrown which is also defined by the framework.

4.4 Implementation for mobile devices

This section describes the implementation of a user interface application for use with the framework described above, developed as part of this thesis. This version of the user interface application was developed for running on devices that use the Android operating system, thus covering both smartphones and tablet devices, and was therefore implemented using the Java programming language and the Android software development kit (Android SDK[2]).

4.4.1 Application components

The user interface application's main purpose is to allow the user to start an AAL service, and to present output and input interactions to the user on instruction by services. The applications architecture represents these two tasks, that are usually executed one after the other.

The application therefore contains two activities, one for letting the user choose and start an AAL service and one for presenting interactions, and one background service. Working with the Android SDK, an activity represents a class that contains definitions of objects to be displayed on the screen, while a service is executed in the background.

To avoid misunderstandings of Android services with the framework-specific AAL services, the full term "AAL service" is used for the latter throughout this section while the former is referred to as "background service".

AAL service chooser activity

The activity that is shown on application startup establishes a connection to the server application and requests a list of all available AAL services from the server application. A list of command buttons is displayed on the screen, each representing one AAL service. By tapping on one of the buttons, the chooser activity sends a command to the server application, requesting it to start that AAL service.

On the UI device, as soon as the chosen AAL service has been started by the server application, the chooser activity starts the background service and is then closed.

Background service

The background service's purpose is to wait for operations received from the AAL service, process these operations by initiating interactions, and report the results back to the AAL service. It is continuously executed in background, until an operation of type `EndOperation` is received.

The first step is to establish a connection with the currently running AAL service. Then, the background service starts an asynchronous task that listens to the connection and waits for operations. This task is running in a separate thread, thus not blocking the application's execution.

[2] `http://developer.android.com/sdk/`

When an operation was received, the next steps depend on the operation's type:

- If an `OutputOperation`, `AlertOutputOperation`, `InputOperation`, `AlertInputOperation`, `GroupOperation`, or `AlertGroupOperation` was received, a new instance of the presentation activity is created and the operation's content is handed over to this activity. The presentation activity is then responsible for processing the desired interactions. In the meantime, the background service waits for additional operations.

- If a `CancelOperation` was received, the current instant of the presentation activity is instructed to cancel the currently active interaction. Also in this case, the background service continues with waiting for further operations.

- If an `EndOperation` was received, the current instant of the presentation activity - if existent - is closed, and the background service stops itself.

In addition to listening for operations sent from the AAL service, the background service continuously waits for results sent from the presentation activity. As mentioned before, most types of operations require a result to be sent back to the AAL service. This is in the responsibility of the presentation activity. Whenever an interaction has been fully processed, the background service receives an appropriate result from the presentation activity and forwards it to the AAL service.

Presentation activity

The presentation activity's task is to process exactly one operation received from the background service (which forwarded it from the AAL service), and report back the results to the background service (which again will forward them to the AAL service). After this task has been fully processed (or has been interrupted by a cancel operation), the activity is closed.

However, in some use cases of a typical AAL service several interactions will be processed successively, meaning that a few moments after reporting the result of one interaction to the background service and closing the presentation activity, it will be started again for processing the next interaction. In this case, the user might experience the effect of short interruptions in the optical application flow in which the smartphones main menu or any other currently running application is shown for the short timespan when no instance of the presentation activity is active. To avoid this effect, the presentation activity waits for three seconds between sending a result to the background service and closing itself. This timespan should be long enough even when using a slow network connection to brigde the timespan between two interactions.

The presentation activity defines two internal methods `getViewForOutput` and `getViewForInput`. Those are responsible for converting data types to Android SDK specific UI widgets to be used in output and input interactions, respectively. Section 4.4.2 explains in detail how these methods work for each data type. How the result of these two methods is used depends of the type of interaction:

- In case of a single output interaction, the widget returned by the `getViewForOutput` method is displayed on the screen in combination with the caption, if a caption value was defined by the original operation, until either the next interaction is to be processed or the current interaction is canceled.

- In case of a single input interaction, the widget returned by the `getViewForInput` method is displayed on the screen in combination with the caption, if a caption value was defined by the original operation, and a confirmation button. The user may use the widget to enter the requested data, and tap on the confirmation button when finished.

  The input is validated automatically while being entered, if the data does not confirm to the formal restrictions (if any were defined by the original operation), the confirmation button is disabled and an icon is displayed indicating that invalid data has been input.

  When tapping on the confirmation button, an `InputResult` object is created including the entered data and sent back to the background service. After three seconds, the activity is closed.

- In case of several simultaneous interactions, the methods `getViewForOutput` and `getViewForInput` are called repeatedly, once for each nested interaction. All returned widgets are displayed at once, in combination with the caption if defined by the original `GroupOperation`, and a confirmation button.

  Depending on the screen size of the UI device, a maximum number of widgets to be displayed at once is defined. If a `GroupOperation` contains more interactions, they are partitioned and shown successively on several screens. In this case, each screen except for the last one contains a forward button instead of the confirmation button.

  When tapping on the confirmation button, a `GroupResult` object is created including the results of all interactions and sent back to the background service. After three seconds, the activity is closed.

4.4.2 Presentation of output and input interactions

The `getViewForOutput` and `getViewForInput` methods are responsible for generating widgets that can directly presented to the user based on the content of output and input interactions. The results of this conversion might differ based on the following criteria:

- Data type
- Interaction type (output or input)
- Formal restrictions
- Semantic information
- Alert type

- Device characteristics

One example for the last criterion, device characteristics, used by the prototype of the mobile user interface application is screen size. On startup of the presentation activity, the constant value `screenCharacteristics` is set to either `DISPLAY_LARGE` or `DISPLAY_SMALL`, depending on the currently used UI device's screen density and resolution. The aim is to distinguish between smartphones and tablet devices. In general, one of two different font sizes is used for all data displayed on the screen, depending on the `screenCharacteristics` value, in order to display information as large as possible, following the guidelines listed in section 3.6.3. In addition, the amount of simultaneous interactions shown at once also depends on this value.

At the same time, the variable `deviceCharacteristics` is set based on the abilities and features of the currently running UI device. This constant may contain the values defined as *Device Characteristics*, as explained in section 4.3.1. Based on this value, for some data types the `getViewForOutput` and `getViewForInput` methods might return completely different widgets.

Finally, for each interaction one of the constant values defined as *Presentation Characteristics*, as explained in section 4.3.1. Also this value might influence the result of the `getViewForOutput` and `getViewForInput` methods.

In the prototype implementation of the mobile user interface application, the alert type criterion does not influence the individual conversion from date type to toolkit-based widget. Instead, when using one of the operation types `AlertOutputOperation`, `AlertInputOperation`, or `AlertGroupOperation`, an alarm sound is played and a vibration alert is started when displaying the final widget on the screen, both of which do not end until the user confirms the interaction, as suggested by the results of the first user workshops (see section 3.6.2 for details).

Text

Output mode: The textual content of the `Text` data type is simply displayed on the screen.

Input mode: A text box is displayed on the screen that contains the textual content of the `Text` data type, if any content was defined. If a maximum length was defined, this is directly implemented by the text box by not allowing to enter more than the given amount of characters. Minimum length and regular expression pattern, if defined, are validated on input. In case of an invalid input a small icon showing an exclamation mark is displayed next to the text box.

Boolean

Output mode: The content of the `Boolean` data type, either `true` or `false`, is converted to a textual value containing either "Yes" or "No" which is displayed on the screen.

Input mode: For using the `Boolean` data type in an input interaction, a check box is used.

Number

Output mode: The numeric of the `Number` data type is formatted to fit the given accuracy and number of decimal places, if defined, and displayed on the screen as text.

Input mode: For the input of numeric data, a special `NumberPicker` widget is used. This widget consists of a text box and two buttons with captions + and -. The value displayed in the text box can be changed directly through a numeric on-screen keyboard that shows only digits and the characters "-" and "," since those are required for the input of negative and decimal numbers.

Alternatively, the value displayed in the text box can be increased or decreased by tapping on one of the buttons. In this case, the current value in increased or decreased by steps that correlate to the accuracy value defined by the `Number` data type. When touching one of the buttons for a longer period of time, the current value is increased or decreased incrementally, until the finger is removed from the button.

The other formal restrictions, if defined, are validated on input. In case of an invalid input a small icon showing an exclamation mark is displayed next to the text box.

Time

Output mode: The content of the `Time` data type is converted to a textual value, matching one of the formats *HH:MM:SS.mmm*, *HH:MM:SS*, or *HH:MM*, and displayed on the screen. The format used depends on the accuracy of the value to be displayed, if it does not encode milliseconds or seconds, the respective parts will be omitted in the output.

Input mode: For the input of a point in time or a period of time, four widgets of type `NumberPicker` are used, encoding hours, minutes, seconds, and milliseconds. Depending on the desired accuracy defined by the `Time` data type, two, three, or all four `NumberPicker` widgets are shown on the screen - if the value shall not encode seconds or milliseconds, those widgets are not displayed. Figure 4.5 shows a screen shot of an input interaction of type `Time` with the widgets for hours, minutes, and seconds being visible.

Minimum and maximum values for all four `NumberPicker` widgets are set automatically. Usually, this means intervals of $[0, 23]$ for hours, $[0, 59]$ for minutes and seconds, and $[0, 999]$ for milliseconds. If a minimum or maximum value was defined as part of the formal restrictions, those intervals are adapted accordingly. The step size of all widgets is set to 1, only if an accuracy value was defined as part of the formal restrictions the step size of the widget that represents the smallest unit (minutes, seconds, or milliseconds) is adapted according to this accuracy value.

When incrementing or decrementing the value of one of the widgets by using the + or - buttons, neighboring widgets are updated automatically: For example, if the value representing minutes is changed from 59 to 0, the value representing hours is automatically incremented.

The input is constantly validated for matching the formal restrictions. If invalid data has been entered, a small icon showing an exclamation mark is displayed next to one of the widgets.

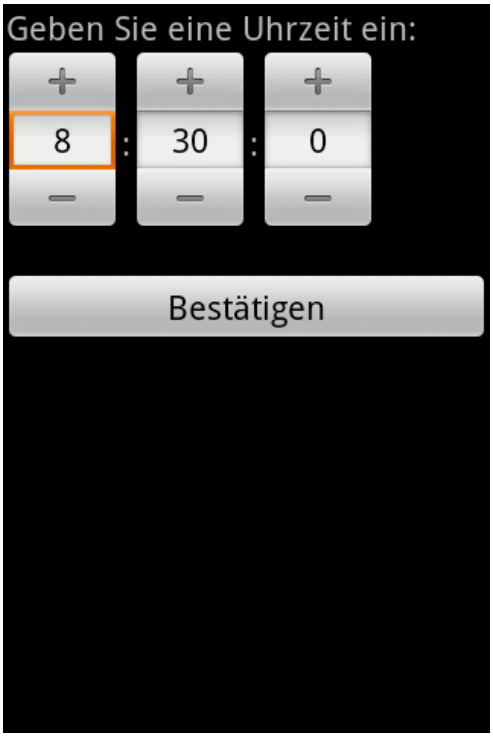


Figure 4.5: Screenshot of the user interface application for Android devices while showing an input interaction requesting data of type `Time`

Date

Output mode: The content of the `Date` data type is converted to a textual value, expressing either year, month and year, or day, month and year, and displayed on the screen. The format used depends on the accuracy of the value to be displayed, if it does not encode the exact day or month, the respective parts will be omitted in the output.

Input mode: For the input of a date, three widgets are combined and displayed on the screen: Two widgets of type `NumberPicker` for specifying day and year, and one drop-down list box for selecting the month. Depending on the desired accuracy defined by the `Date` data type, only some of those widgets are shown on the screen - if the value shall not encode the exact day or month, the respective widgets for selecting day and month are not displayed. Figure 4.6 shows a screenshot of an input interaction of type `Date` with all widgets for day, month, and year being visible.

The minimum and maximum value for the `NumberPicker` widget that represents the day is set automatically to the interval $[1, 31]$. If a minimum or maximum value was defined as part of the formal restrictions, this interval is adapted accordingly. The step size of all widgets is set to 1, only if an accuracy value was defined as part of the formal restrictions the step size of the widget that represents the day is adapted according to this accuracy value.

Figure 4.6: Screenshot of the user interface application for Android devices while showing an input interaction requesting data of type `Date`

The input is constantly validated for matching the formal restrictions and for being a correct date (for example, a value of 31 as day is not appropriate for all months). If invalid data has been entered, a small icon showing an exclamation mark is displayed next to one of the widgets.

GPS

Output mode: The presentation of geographic data depends of the space available on the screen. The `getViewForOutput` method checks whether the presentation characteristics of the currently active output interaction are defined by the flag `PRESENTATION_SINGLE` or `PRESENTATION_GROUP`. In the first case, the current interaction is being processed as single output interaction, meaning that the full screen can be used for presenting the content of the `GPS` data type. In the second case, the current output interaction is one of several simultaneous interactions, meaning that at least one other data item is to be presented on the screen at the same time, thus reducing the screen space that may be occupied by the `GPS` data item.

In both cases, the given GPS coordinates are converted to a street address using the Google geocoding API[3], and this address is displayed as text on the screen.

[3] `https://developers.google.com/maps/documentation/geocoding/`

If the full screen can be used, a map view is displayed beneath the textual street address, on which the exact location is marked by a pin. If not, a command button is displayed in combination with the street address that can be tapped on by the user in order to open a map view showing the given location using the full screen. Figure 4.7 shows two screenshots illustrating those two solutions.

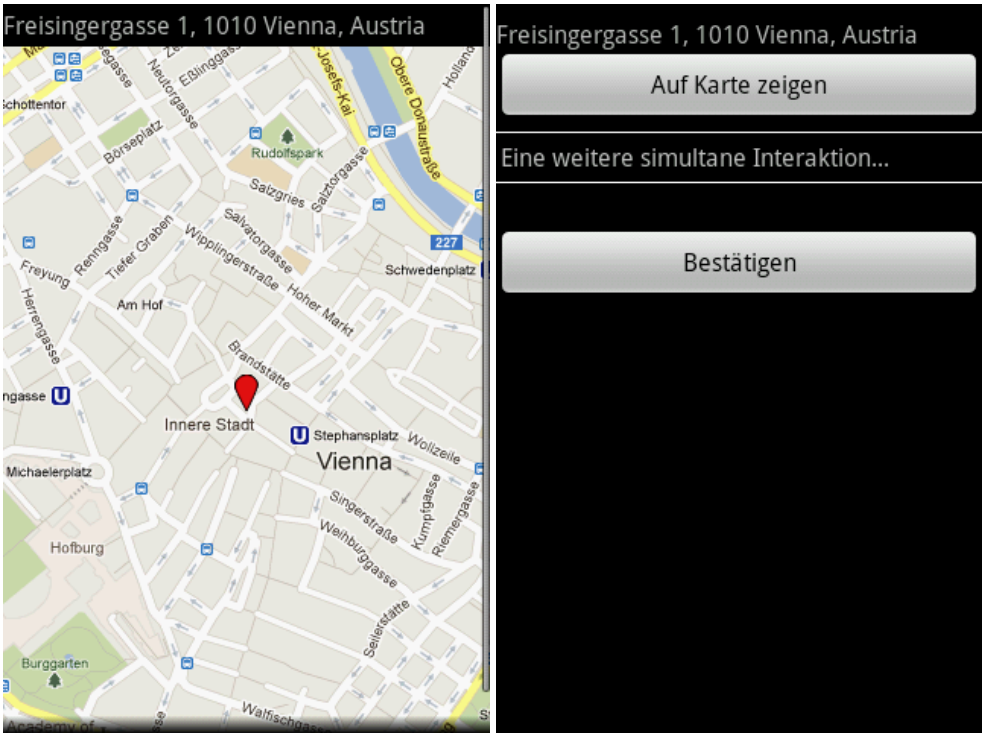


Figure 4.7: Screenshots of the user interface application for Android devices while showing output interactions presenting data of type `Date`, either as single output operation (left) or as one of several simultaneous interactions (right)

In both cases, an instance of Google Maps[4] is used for the map view. This product has been chosen due to its seamless integration into the Android operating system.

Input mode: If the semantic flag `IS_CURRENT_POSITION` is set, the user might not even be asked to input a location. If the currently used UI device features a location retrieval sensor, if this sensor is turned on, if the access to this sensor has been permitted by the user, and if a GPS signal can be received, the user interface application retrieves the UI device's current location and returns that as result.

If one of these requirements does not apply or if the flag `IS_CURRENT_POSITION` is not set, the user is requested to enter a location. This is done through a text box in which the user

[4] `https://developers.google.com/maps/`

can enter a street address. This address is then converted to GPS coordinates using the Google geocoding API.

SingleChoice

Output mode: The selected item is displayed on the screen as text.

Input mode: The selection of one item from a list is realized in two different ways, depending on the space available on the screen (expressed through the presentation characteristics of the currently active output interaction which are defined by the flag `PRESENTATION_SINGLE` or `PRESENTATION_GROUP`). If the `SingleChoice` data type is processed as content of a single input interactions and therefore the full screen can be used, a list of radio buttons is displayed, one of which may be selected by the user through tapping on it.

Otherwise, if a group of simultaneous interactions are being processed and therefore the available screen space is limited, a drop-down list is used that can be opened by tapping on it and then shows the full list of selectable items. Figure 4.8 shows two screenshots illustrating those two solutions.

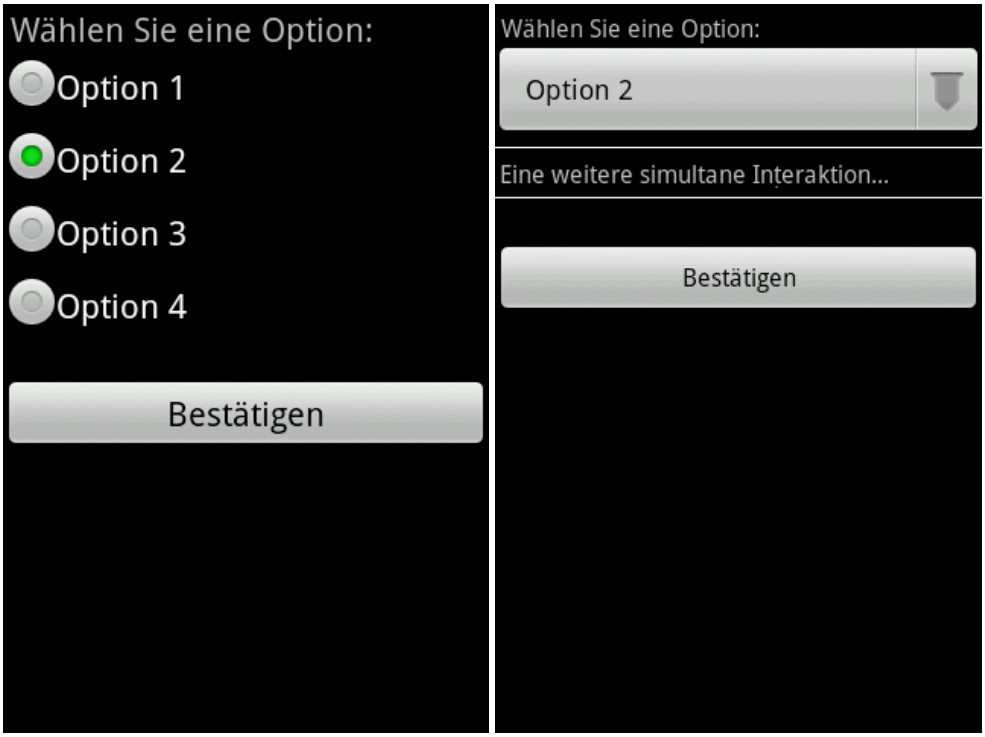


Figure 4.8: Screenshots of the user interface application for Android devices while showing input interactions requesting data of type `SingleChoice`, either as single input operation (left) or as one of several simultaneous interactions (right)

MultipleChoice

Output mode: All selected items are displayed on the screen as text, either separated by a comma or each one in a new line, depending on the space available on the screen. This is decided based on the presentation characteristics of the currently active output interaction, whether the flag `PRESENTATION_SINGLE` or `PRESENTATION_GROUP` is set.

Input mode: The selection of one item from a list is realized in two different ways, depending on the space available on the screen (expressed through the presentation characteristics of the currently active output interaction which are defined by the flag `PRESENTATION_SINGLE` or `PRESENTATION_GROUP`). If the `MultipleChoice` data type is processed as content of a single input interactions and therefore the full screen can be used, a list of checkboxes is displayed, which may be selected and de-selected by the user through tapping on one of them.

Otherwise, if a group of simultaneous interactions are being processed and therefore the available screen space is limited, a drop-down list is used that can be opened by tapping on it and then shows the full list of selectable items. Figure 4.9 shows two screenshots illustrating those two solutions.

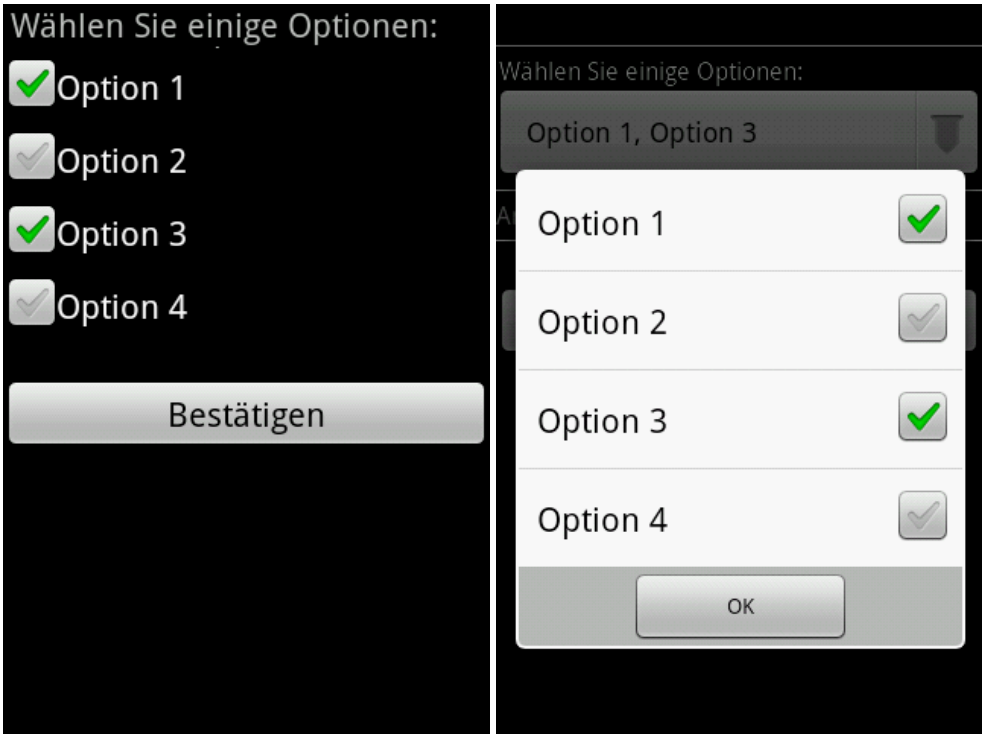


Figure 4.9: Screenshots of the user interface application for Android devices while showing input interactions requesting data of type `MultipleChoice`, either as single input operation (left) or as one of several simultaneous interactions (right)

PhoneNumber

Output mode: The telephone number is displayed on the screen as text, including all blank spaces and delimiting characters in order to be readable best by humans. In addition, it is arranged as a command button, meaning that it can be tapped on to directly call that number, if the currently used UI device allows this.

Input mode: If the semantic flag `IS_USERS_NUMBER` is set, the user might not even be asked to input his phone number. If the currently used UI device is able to send and receive phone calls and if access to the UI device's phone number is permitted, the user interface application retrieves the number automatically and returns that as result.

If one of these requirements does not apply or if the flag `IS_USERS_NUMBER` is not set, the user is requested to enter a phone number in a text box. For entering characters in this text box, a numeric on-screen keyboard is used that shows only digits and certain delimiting characters. If an area code was defined as formal restriction, this code is displayed next to the text box. The user is therefore restricted to enter only phone numbers belonging to this exact area, and no misunderstanding can occur whether the phone number to be entered must include an area code or not.

Minimum and maximum length, if defined, are validated on input. In case of an invalid input a small icon showing an exclamation mark is displayed next to the text box. The same applies if the data entered does not match the typical format of a telephone number.

If the semantic flag `IS_KNOWN_BY_USER` is set and the currently used UI device provides a contacts application, in addition to the text box a command button is displayed that allows opening this contacts application. In this case, a contact may be chosen there and his or her telephone number is used as result, if it matches the formal restrictions and belongs to the area code specified by the `PhoneNumber` data type.

Figure 4.10 shows a screenshot of an input interaction requesting data of type `PhoneNumber`, illustrating both the area code and the command button for selecting an existing contact's phone number.

EmailAddress

Output mode: The e-mail address is displayed on the screen as text. In addition, it is arranged as a command button, meaning that it can be tapped on to directly compose an e-mail and send it to the address, if the currently used UI device allows the composition and sending of e-mail messages.

Input mode: For entering an e-mail address, a text box is displayed on the screen. If a domain name was defined as formal restriction, this domain is displayed next to the text box. The user is therefore restricted to enter addresses belonging to this domain.

Minimum and maximum length, if defined, are validated on input. In case of an invalid input a small icon showing an exclamation mark is displayed next to the text box. The same applies if the data entered does not match the typical format of an e-mail address.

Figure 4.10: Screenshot of the user interface application for Android devices while showing an input interaction requesting data of type `PhoneNumber`

If the semantic flag `IS_KNOWN_BY_USER` is set and the currently used UI device provides a contacts application, in addition to the text box a command button is displayed that allows opening this contacts application. In this case, a contact may be chosen there and his or her e-mail address is used as result, if it matches the formal restrictions and belongs to the domain specified by the `EmailAddress` data type.

Figure 4.11 shows a screenshot of an input interaction requesting data of type `EmailAddress`, illustrating both the area code and the command button for selecting an existing contact's address.

Command

Output mode: The data type `Command` may only be used in input interactions.

Input mode: The data type `Command` is represented as a command button that can be tapped on by the user.

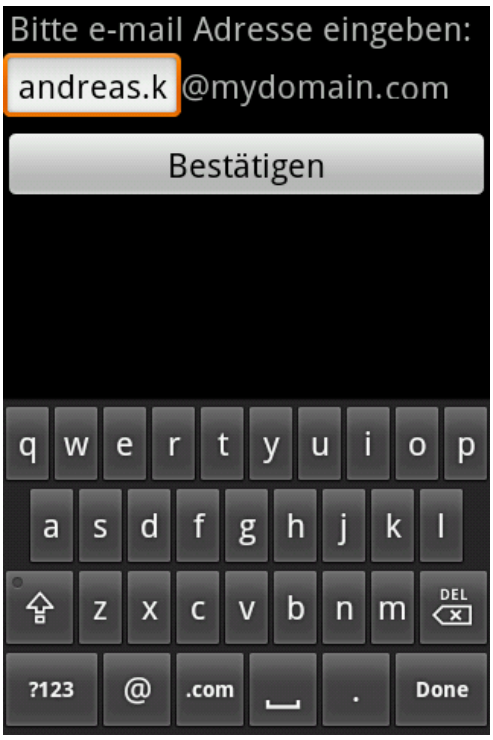


Figure 4.11: Screenshot of the user interface application for Android devices while showing an input interaction requesting data of type `EmailAddress`

4.5 Potential entensions

The prototype version of the user interface generation system implemented as part of this thesis is intended as a proof of concept for basic testing purposes rather than a solution for use in productive environments. To further improve this first prototype, several ways of extending and adapting the implementation are imaginable. This section suggests a three-tiered approach of extending the use-cases of the user interface generation system that could be accomplished in future versions.

4.5.1 Enhancing data types and their presentation

The first step focuses on extending and enhancing the prototype versions of the framework and the user interface application for mobile devices. This includes the extension of the set of predefined data types, the enhancement of flexibility when presenting these data types on UI devices, and the integration of specific features offered by UI devices which might contribute to additional convenience for the user.

As mentioned before, the set of data types defined by the framework shall resemble a typical user's understanding of which different types of information exist. In addition, the information encoded by a data type must be sufficiently standardized to ensure that any value of this type of information may be encoded using the data type.

One example of an additional data type that could be introduced in future versions of the framework is one that encodes street addresses. Although a street address consists of several parts, such as street, city, country, ZIP code, etc, and the exact combination of those part might differ in various countries and societies, it should be possible to create an object that holds all those parts and defines their relation to each other. The exact presentation of the content of such a data type as well as the way of requesting such data from the user can differ, depending on the language and country settings of the UI device used.

Similar to the `GPS` data type present in the current version of the framework, map views and geocoding systems can be used to provide different presentation mechanisms of a potential `StreetAddress` data type. For example, by converting a given street address to GPS coordinates using a third-party geocoding solution, it is possible to show the exact location on a map, as illustrated by the existing `GPS` data type. Vice versa, for using the new data type in input interactions, a UI device featuring graphical user interfaces might ask the user to choose a location on a map and then convert the coordinates of this exact point to a street address.

A different approach that also aims at improving the predefined set of data types is to enhance existing data types instead of introducing new ones. This can be shown by taking the existing `SingleChoice` and `MultipleChoice` data types as examples: In the current version, they are restricted to presenting a list of textual values, one or several of which may be chosen by the user. Extending the permitted format of those list items from simple textual data to more complex objects would increase service developers' flexibility.

In future versions of the framework, the `SingleChoice` and `MultipleChoice` data types might for example allow a list of items that must be of any of the predefined data types,

instead of restricting them to be of type `String`. The presentation of each of the list items could then follow the guidelines each user interface application defines for the presentation of a single instance of the respective prototype. Since each user interface application must define guidelines for the presentation for each of the predefined data types anyway, the effort for extending the `SingleChoice` and `MultipleChoice` data types should be rather low.

One example for including specific features certain UI devices provide and using them as part of the user interface applications is already realized by the existing data type `PhoneNumber`: If the UI device currently used provides a contacts application including telephone numbers, it may be used for selecting one person's phone number instead of entering the number manually. Since many mobile devices which can be used as UI devices provide a calendar application, the data provided by this calendar application could be integrated in a similar way.

One way of such an integration of the user's personal schedule is through the combination of the existing `Date` and `Time` data types. A data type that encodes an exact date including the time of day could be used to check its content against the user's personal appointments stored in the UI device's calendar application. When presenting the content of such a data type through the user interface application, an additional message might be displayed to the user if the given date conflicts with an existing personal appointment. Also when requesting the input of a date and time of day, the user interface application could instantly notify the user if he or she entered a date that conflicts with a private appointment.

Finally, another approach for enhancing the existing user interface application is to distinguish different types of presentation for each data type, depending on the current UI device's characteristics. This is already included in the prototype version of the user interface application for mobile devices, since it distinguishes smartphones from tablet devices by measuring screen density and resolution. By improving the accuracy of this measurement and introducing additional degrees of granularity beyond the existing flags `DISPLAY_LARGE` or `DISPLAY_SMALL`, differently sized smartphones and tablet devices could be distinguished from each other, thus ensuring the best way of presentation for each of those devices.

4.5.2 Increasing the set of applicable devices and modalities

Apart from adapting the existing prototype versions of the framework and the user interface application for Android devices, an important step towards extending the use-cases of the user interface generation system is the development of user interface applications for additional device types and modalities.

First of all, this concerns other operating systems for mobile devices: User interface application versions for the most common operating systems running on smartphones and tablet devices should be developed. This might include, for example, the Apple iOS[5] operating system which is used by iPhone smartphones, iPad tablets and the MP3 player series iPod touch, and the Microsoft Windows RT operating system which is one edition of the upcoming Microsoft

[5] `http://www.apple.com/ios/`

Windows 8[6] system designed specifically for use on smartphones and tablet devices.

To allow additional use-cases of the user interface generation system, the support of other UI devices apart from mobile devices is most important, because it allows users to choose which types of device they want to use for executing a certain AAL service. This concerns both devices that provide graphical output and input interactions, mainly including personal computers and laptop computers, and other types of devices. One example for the latter is a TV set, which may provide graphical output but requires either voice-based input or input through a basic remote control.

One way to reduce the effort of implementing specific user interface applications for each device type is the usage of HTML-based solutions for those UI devices that are not yet supported by a specific user interface application version. For details on HTML, see section 2.2.15. The basic idea of this approach is the automatic generation of a standardized user interface that can be interpreted by as many devices as possible. HTML is used for describing this standardized user interface because it is supported by a variety of potential UI devices nowadays.

This approach requires the introduction of a wrapper solution that resides on the service hardware layer but acts as user interface application. This wrapper software receives commands from AAL services and creates HTML pages resembling the interactions encoded by the commands. The HTML pages are then presented to the user by any device that is able to interpret HTML code. This includes graphical UI devices such as mobile phones, personal computers, and TV sets which display the contents of each HTML page to the user, but also voice-based input/output systems that present the HTML content by using text-to-speech converters. In case of input interactions, the wrapper then receives the entered data, converts it to the required data type and forwards it to the AAL service.

Such an approach might not produce the best imaginable user interfaces because it does not take into account device-specific features and characteristics as specific user interface application versions would do, but it can act as an auxiliary solution that enables the use of special UI devices which otherwise would not be supported by the user interface generation system. This improves the user's flexibility in choosing a specific UI device for each AAL service, which is a requirement for the third step presented in the following section.

4.5.3 Automatic user interface device selection

With the possibility of additional types of devices being available as UI devices, some users might wish to be able to change from one UI device to another while executing an AAL service, instead of being restricted to choosing the desired UI device before starting a service.

In fact, some of the user interface description languages presented in chapter 2 include solutions that offer this possibility, allowing the user to switch from one UI device to another while executing a remote service. The user interface generation system developed as part of this thesis can be adapted to fulfill a similar use-case by introducing a middleware component to the basic system architecture.

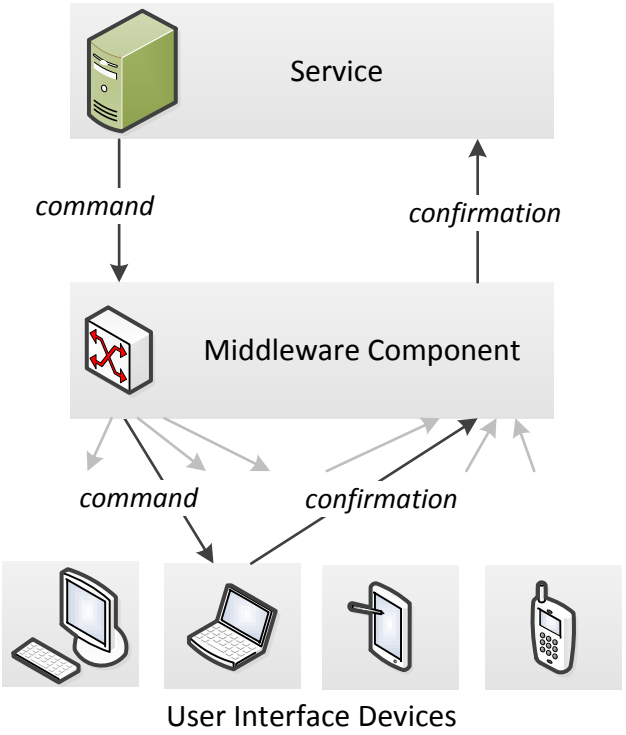
[6] `http://windows.microsoft.com/en-US/windows-8/release-preview/`

The newly introduced middleware component resides on the service hardware layer and acts as additional node in the communication between service and user interface application. Since more than one user interface application is involved at the same time - one for each UI device available in the system - the service does not communicate directly with the user interface application by sending commands and receiving confirmations. Instead it sends commands to and expects confirmations from the middleware component.

It is important to notice that although several UI devices can be present in the course of an AAL service execution using the approach of introducing a middleware component, only one of those UI devices may be active at a certain point of time. The user chooses which device is currently active and therefore shall be used for the following output or input interaction.

The middleware component knows which UI device is currently active and forwards the commands received from the service to this exact UI device. The user interface application running on the respective UI device processes the commands as usual and sends confirmations back to the middleware component which forwards them to the service. Figure 4.12 illustrates the collaboration of service, middleware, and user interface application, showing the changed communication between service and user interface application compared to the original concept explained in section 4.1.

Figure 4.12: Communication between service and several UI devices with intermediary middleware component



The effort of adapting the user interface generation system to follow this approach is rather low, since neither services nor user interface applications need to be modified. They only need

to ensure that a connection to the middleware component is established, instead of a direct connection between service and user interface application.

CHAPTER 5

Evaluation

5.1 Analysis

The prototype of the user interface generation system presented in the previous chapter has been implemented with focus on the intended use as part of an AAL environment. Is it based on the user interaction mechanisms and guidelines established in cooperation with potential users listed in section 3.6.3. Since the prototype acts only as the first version which shall be further enhanced and extended, a comprehensive evaluation of this prototype version has been conducted as part of this thesis.

This section formally analyzes the system's architecture as described in sections 4.1 and 4.2 as well as the actual implementation, summarizes advantages and drawbacks of applying the chosen theoretical concepts to a practical solution, and compares the first version of the final product - the prototype that has been developed - to the other solutions presented in chapter 2.

5.1.1 Strengths

The system's architecture, as described in section 4.2, forces both service providers and user interface application developers to follow strict rules and procedures. This causes the following benefits in every-day use-cases:

- By concentrating the core functionalities of an AAL service in a simple Java application that acts as central control unit, the necessity of communication between different components and devices is reduced to a minimum, thus also reducing the risk of problems occurring in the collaboration of different components.
- At the same time, however, an AAL service's internal procedures are not limited by the system in any way, thus enabling service providers to develop both very simple and highly complex service applications. AAL services may also make unlimited use of supplemental back-end devices, such as web servers, database systems, storage devices, etc., since the collaboration of services with these end devices is also not constrained in any way.
- Specialized user interface applications running on UI devices are responsible for the direct communication with users, which ensures maximum quality of output and input interactions.
- The approach of outsourcing user interaction tasks to UI devices also enables AAL services to implicitly taking into account a user's preferences when presenting information or requesting data for input.
- In addition, it allows the realization of a system similar to the migratory approach for switching between UI devices in the course of an AAL service execution suggested by the authors of [34], as stated in section 2.2.10. This can be accomplished making minor changes to the architecture of the user interface generation system and is a unique feature, since only a few of the various User Interaction Description Languages analyzed in chapter 2 provide similar options.

- The restriction to use only data types out of the list predefined by the framework ensures seamless communication between AAL services and user interface applications. It also allows the development and integration of additional services and user interface applications without the need of adapting the framework and the system's architecture in the future.

- The definition of each of the predefined data types is completely independent of modalities and does not specify which user interface widgets to be used, but nevertheless allows specifying detailed semantic information about the contents of a data item.

In addition, the concrete implementation of the components that comprise the architecture mentioned above was developed in order to follow the guidelines established in section 3.6.3 whenever possible, resulting in the following advantages of the system when compared to similar solutions:

- The implementation of the user interface application for Android-based devices following the guidelines established together with potential users as described in section 3.6.3 whenever possible ensures maximum accessibility for the prospective user group of the user interface generation system.

- In addition, the user interface application is based on the common themes and UI widgets offered by the Android operating system, in order to ensure a familiar user interface experience for users already familiar with Android devices.

- One of the guidelines for implementation established in section 3.6.3 suggests to maximize the action area of UI widgets that expect to be tapped on by users. The implementation of the user interface application for mobile devices realizes this by displaying command buttons as large as possible, always filling the full width of the screen, to ease tapping on these buttons.

- The same guideline states that all contents apart from operational widgets should also be displayed as large as possible, which is fulfilled by establishing different font sizes depending on the size and resolution of the UI device's screen for displaying textual content.

- While the above-mentioned font sizes used for displaying are set according to the UI device's display characteristics, one constant font size is used throughout the whole execution of an AAL service in order not to confuse users and suggest different degrees of importance through using differently sized text elements.

- The framework which includes implementations of the most important functions to be used in an AAL service's application flow allows easy and convenient development of both services and user interface applications.

5.1.2 Shortcomings

On the other hand, the fixed structure may be rather inflexible, limiting service providers' and user interface application developers' possibilities, resulting in the following disadvantages:

- The restriction to use only data types out of the list predefined by the framework limits the possibilities of service providers, since they must express each data item to be presented to or requested from the user as one of these data types. It is not possible to define custom objects without extending the framework.

- In addition, since the set of predefined data types are chosen due to being especially user-friendly instead of due to their technical characteristics, the memory consumption and therefore also the time necessary to transfer data between services and user interface applications is slightly higher than when using data types embedded into the programming language used.

- One serious problem in consistency with common Android user interaction patterns is the missing integration of the back button. This is a hardware button present on all Android-based devices that is commonly used when interacting with the device. Pressing the hardware back button usually results in the current screen being closed, and the currently running application returning to the screen previously displayed. It can be used to review and change settings specified using an earlier screen.

  In the user interface application for mobile devices, each screen is confirmed by tapping on the confirmation button. The screen is then closed and the results are reported to the AAL service. This characteristic feature of the system's architecture means that data which was presented to the user or input by the user at an earlier point in time cannot be reviewed or even changed, because the UI device has no access to this data anymore.

  Therefore, the back button does not work when relying on this architecture. In fact, pressing the back button while executing the prototype version of the user interface application has no effect, which might seriously confuse users who are accustomed to working with the Android operating system.

- Another drawback is the fact that service providers are not offered any possibilities to influence the presentation of their services on any UI device. Some service providers may want to display their logo or company name when executing one of their AAL services, which is not offered by the current version of the system.

The following shortcomings of the concrete implementation of service interface, framework, and user interface application for mobile devices can be identified:

- One example for the restriction to using only the predefined data types is the presentation of a list of items using one of the data types `SingleChoice` or `MultipleChoice`. They allow the presentation of textual data items, but all text elements are displayed using the same font size, style, and color. Service providers might wish to emphasize parts of the

contents of each data item that are more important by using bold font styles, for example, which is not possible using the current system version.

- The implementation of the user interface application for mobile devices follows the guidelines established together with potential users as described in section 3.6.3 whenever possible, however some of the suggestions could not be considered and need therefore be taken into account by service providers while implementing AAL service applications.

  One example is the suggested use of a paper form metaphor for designing input forms. In the course of the implementation, it was considered too limiting to completely disallow preselected items in a group of several check boxes. This guideline needs to be forwarded to service providers and taken care of while designing service applications.

5.1.3 Comparison with other User Interaction Description Languages

This section analyzes the prototype of the user interface generation system based on the criteria defined in section 2.1. Due to the intended application in the field of Ambient Assisted Living and the system being a first prototype version, criteria 1 and 2 are the most interesting ones. The other criteria are mentioned for reasons of completeness.

Finally, a comparison chart including the characteristics of all UIDLs analyzed in chapter 2 and the system presented in chapter 4 is shown in table 5.1.

1. **Level of abstraction**

   The communication between AAL services and UI devices is restricted to contain only predefined data types, and those data types are implemented as to not include any modality- or toolkit-specific details while nevertheless defining a data type's nature as detailed as possible. Therefore in theory all imaginable front end devices using all different types of modalities can be used as UI devices, implying a high level of abstraction. In practice, of course, a user interface application must be implemented for each of those potential UI devices. To support also devices for which no user interface application is available, the approach using an auxiliary HTML wrapper, as described in section 4.5.2, could be used.

2. **Adaptability**

   By outsourcing the interaction with users to UI devices the user is accustomed to and has already adjusted to fulfill his or her needs, maximum accessibility is guaranteed. The adaptation of user interfaces to different environmental influences is also in the responsibility of the UI devices, therefore the criterion of context-awareness can not be evaluated for the overall system.

   The migratory solution for switching between different UI devices as presented in section 4.5.3 is one approach towards use-case awareness, since it would allow users to choose which device to use for controlling different parts of an AAL service instead of being bound to execute the whole service from the same UI device. However, an in-depth realization of the concept of use-case awareness, including the presentation of different functionalities on different devices, is not provided by the current prototype version.

3. **Openness**

   All information contained in this thesis is publicly available, and the prototype implementation developed as part of this thesis may be used for research purposes.

4. **Organizational background**

   The user interface generation system presented was developed as part of a diploma thesis at the Vienna University of Technology.

5. **Status**

   This thesis presents the first prototype version of the user interface generation system, including two example AAL services and one exemplary user interface application. In addition, a detailed evaluation of this first version is documented. Suggestions for potential adaptations and extensions of the present system to be realized in future versions are discussed in section 4.5.

6. **Number of implementations**

   The first prototype version includes one user interface application, implemented for mobile devices.

7. **Number of supported target platforms**

   The prototype implementation of the first user interface application works on mobile devices - including smartphones and tablet devices - running the Android operating system.

	level of abstraction	adaptability			
		accessibility	context-awareness	use-case awareness	
AAIML	high	unknown	unknown	yes	
PreT	rather high	medium	no	no	
XIML	rather high	yes	yes	yes	
XISL	medium	no	no	no	
WSDL	very high	N/A	N/A	N/A	
WSXL	high	no	no	no	
UsiXML	rather high	yes	yes	yes	
UIML	medium	prepared	prepared	prepared	
DISL	rather high	no	yes	yes	
MARIA XML	high	prepared	through migrational UI approach	prepared	
XAML	low	no	no	N/A	
XUL	low	no	partly	no	
MXML	low	no	no	no	
VoiceXML	rather low	no	no	N/A	
HTML	low	through browser	no	through CSS	
UI Generation System	high	yes	through UI devices	partly (through extension)	

Table 5.1: Comparison chart of all analysed User Interaction Description Languages, including the system developed as part of this thesis (*UI Generation System*)

	openness	organizational background	status	implementations	supported target platforms
	low	standardization organization	discontinued	5	5
	medium	international consortium	approved as international standard	> 5	2
	high	industry	latest version from 2002	at least 2	2
	high	university	latest version from 2003	at least 3	3
	high	open standards community	W3C recommendation	many	all
	high	industry	latest version from 2002	at least 2	unknown
	high	research organization	latest version from 2007	many	2
	high	standardization organization	latest version from 2009	many	at least 4
	low	university	latest version from 2006	unknown	unknown
	medium	research organization	latest version from 2011	unknown	at least 3
	high	industry	under development	4	3
	high	open source community	under development	1	1
	medium	industry	under development	2	3
	high	open standards community	latest version from 2004	many	1
	high	open standards community	under development	many	5
	high	university	prototype	1	1

5.2 Example services for evaluation

In addition to the formal analysis conducted in the previous section, practical tests were performed involving potential users of the user interface generation system in order to evaluate the system's practical performance in typical use cases.

In preparation for these user tests, the two example services defined is section 3.2 were implemented using the framework presented in chapter 4. This section describes the implementation of these two services as well as their presentation on a typical Android user interface device.

Both example services are simple Java applications that are started by the server application. They indicate that they act as AAL service applications by implementing the public `Service` interface defined by the framework. This implies that they also implement the three overloaded `callback` methods which are called automatically on incoming `Result` objects.

There exists one `callback` method for each type of `Result`, since each result includes different contents and requires different reactions of the AAL service:

- The version expecting an object of type `String` as parameter is used for receiving confirmations that an output interaction, represented by an operation of one of the types `OutputOperation` or `AlertOutputOperation`, has successfully been presented to the user. No information beyond the fact that the information has been output to the user is to be conveyed by such a confirmation, therefore no parameters apart from the `String` object, which represents the original operation's identifier, are necessary.

- The version expecting a result of type `InputResult` is used for receiving data that has been requested from the user as part of an input interaction by sending an operation of one of the types `InputOperation` or `AlertInputOperation`. Apart from the identifier of the operation originally sent, the `InputResult` object contains the actual data entered by the user, contained in an object of any of the predefined data types.

- The version expecting a result of type `GroupResult` is used for receiving all the results of several interactions that were carried out simultaneously. Therefore it can be seen as counterpiece to the `GroupOperation` and `AlertGroupOperation` types of operations. Apart from the identifier of the group operation originally sent, the `GroupResult` object contains the single identifiers of each of the simultaneous operations that were contained in the original group operation, and in case some of those simultaneous operations initiated input interactions also the respective data entered by the user, contained in an object of any of the predefined data types.

All three methods are called by the `Connection` class that handles a service's communication with the user interface application when a result from the currently active user interface application arrives. The `Connection` class is also responsible for forwarding the actual contents of the result received to the service, which is done through the parameters of the `callback` methods.

Most AAL services consist of more than three interactions, therefore each of the `callback` methods may be called more than once. The `callback` implementations of both example services consist of a case distinction that checks for each incoming result the identifier of the related operation and decides how to proceed.

Since the two example services are intended for testing purposes and they present only exemplary data, no communication with external sensors, storage devices, or servers is necessary. Therefore in most cases an incoming result is processed by locally storing the contents of the `Result` object and sending the next operation.

Apart from the implementations of the `callback` methods, the service classes only contain a constructor method that is responsible for the basic setup necessary for sending operations and receiving results. The first step after an AAL service has been started is to establish a connection to the user interface application. This is done by calling the `connect` method that is a member of the `Connection` class. After this step has been successfully completed, an instance of the `Connection` class can be retrieved and used for sending operations throughout the execution of the service application.

After connecting to the user interface application, the constructor method starts the application flow by constructing and sending the first operation to the user interface application. Afterwards, the service waits for the result of this operation. When the `Connection` class receives the result, it automatically calls the service's respective `callback` method and forwards the `Result` object, or the identifier of the original operation in case of an `OutputResult`.

As an alternative to waiting for an unpredictable amount of time until receiving the expected result, a service might use a timer to wait only for a certain period of time. If the result was not received until then, it sends a `CancelOperation` to the user interface application, indicating that it will no longer wait for the result. The exact implementation of this timer is up to the service application. However, the two example services do not make use of this possibility.

Constructing an operation is done as follows: An instance of one of the predefined data type classes that implement the `Type` interface is created and filled with data. The content of the data type object must be defined, otherwise the default content (specified by each data type through the `DEFAULT_VALUE` constant) is used. In addition, formal restrictions and semantic information may be defined, if supported by the data type. Then an object of any of the predefined `Operation` types is created, and unique identifier and content are set. As content, the previously created data type object is passed to the operation object. In addition, an optional text value may be defined that is used as caption. This operation object is finally sent to the user interface application by passing it to the `send` method offered by the `Connection` class.

For constructing group operations used to initiate simultaneous interactions, the previously described steps are repeated incrementally: First, the single operations representing the interactions to be carried out simultaneously are constructed individually, including their content data types. Then, they are all added to an object of one of the types `GroupOperation` or `AlertGroupOperation` by passing them to the `addOperation` method defined by these operation types.

After the last expected result has been received and no further operations will be sent because the AAL service has completed the task it was started for, the connection to the user interface application must be closed by calling the `close` method offered by the `Connection` class.

This method internally creates an object of type `EndOperation` and sends it to the user interface application to indicate that all user interface screens may be closed and the background service which waits for incoming operations can be stopped.

5.2.1 Doctor's appointment service

The first step of asking the user to input search criteria for finding an appropriate doctor is realized as group of two simultaneous interactions: The doctor's field of specialization is represented by an interaction of type `SingleChoice` that contains a hard-coded list of different specializations, one of which may be chosen by the user. The selection of one or several types of insurance accepted by the doctor is represented by an interaction of type `MultipleChoice`.

On a mobile device, these two interactions are presented at the same time as drop-down list widgets that can be opened by tapping on them. Figure 5.1 shows a screenshot of the user interface application for mobile devices interpreting this first group operation.

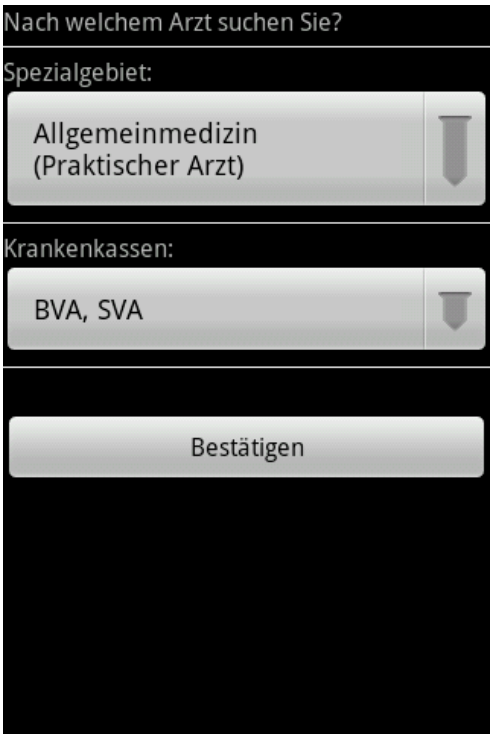


Figure 5.1: Screenshot of the first phase of the doctor's appointment service requesting search criteria, rendered by the user interface application for mobile devices

In the following step, several doctors are presented to the user that fulfill the criteria, and the user may select one of them to make an appointment. This interaction is realized as a single input interaction of type `SingleChoice`. Since the service is used only for testing purposes, a hard-coded list of doctors is used as content of the `SingleChoice` data type instead of

searching an external list of doctors for the given criteria. This list contains each doctor's name, the street address, and the distance to the user's current location.

Also the distance displayed is a fixed value, while in a productive version it should be calculated from the doctor's street address and the user's real location which might be retrieved using an input interaction of type `GPS` with the semantic flag `IS_CURRENT_POSITION` set.

Since the `SingleChoice` interaction representing this phase of the service is presented on its own, the user interface application for mobile devices uses a list of radio buttons, as illustrated in figure 5.2.

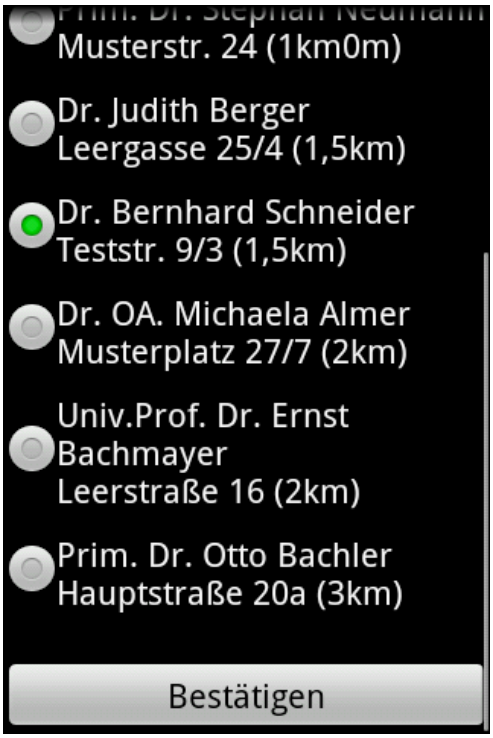


Figure 5.2: Screenshot of the first phase of the doctor's appointment service offering several doctors for selection, rendered by the user interface application for mobile devices

In the second part of the doctor's appointment service, it is possible to directly book an appointment at the previously selected doctor's office. To do so, the name and address of the selected doctor are stored internally by the service application as soon as it is received as content of an `InputResult` object. Then, the user is asked for the desired date of the appointment.

Since the test persons taking place in the first user workshops stated that providing more options than the simple selection of a certain day is desirable, but a high amount of options might be confusing to the user, this step was realized in the example service implementation by asking the user to input a day and the time of day. For selecting the time of day, the user may choose whether the appointment shall take place before or after noon, instead of using a difficult interaction scenario for defining distinct periods of time as in the mockups used for the first user

workshops.

Again, a group interaction is used for this step, including an input interaction of type `Date` for selecting the appointment date, and an input interaction of type `MultipleChoice` for choosing the time of day, because the user might either choose ”before noon”, ”after noon”, or both if the time of day doesn't matter. For the `Date` data type used for the first input interaction, minimum and maximum values are defined in order to allow only the selection of dates later than the current day and in the next three months. The `MultipleChoice` data type used for the second input interaction specifies that at least one and not more than two values may be chosen.

Since both interactions are presented to the user simultaneously on a mobile device, the `MultipleChoice` interaction is realized by the user interface application as drop-down list. Figure 5.3 shows two screenshots of this phase of the doctor's appointment service on a mobile device: The first screenshot shows the opened drop-down list for selecting the desired time of day, while the second one shows the full screen with an invalid date selected which is indicated by the exclamation mark displayed next to the widget for choosing days.

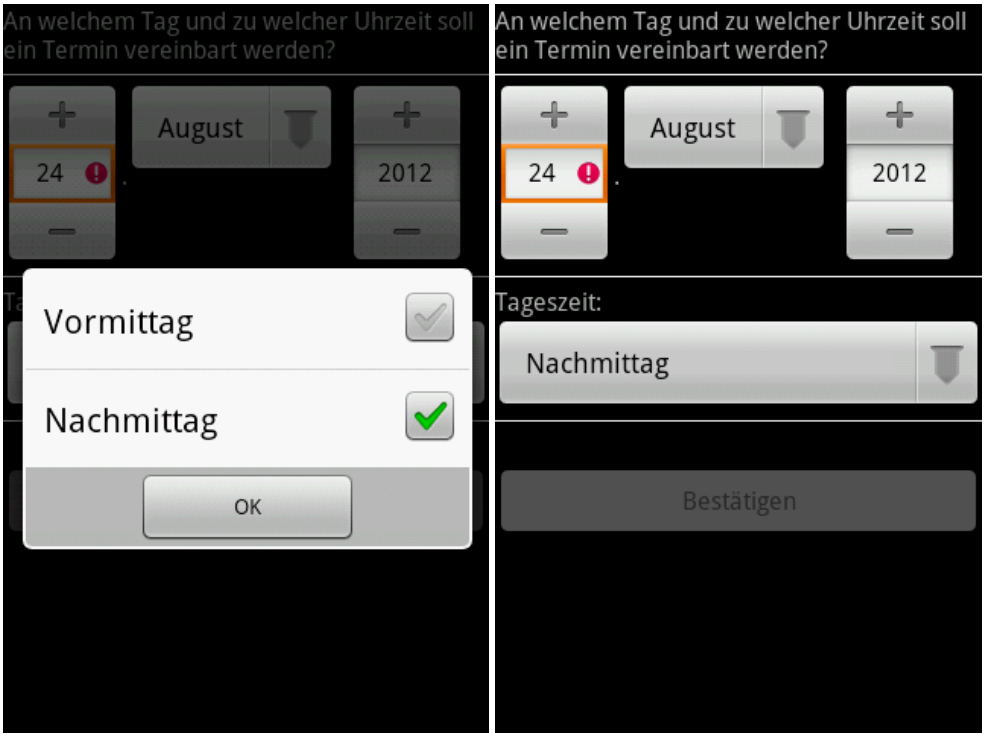


Figure 5.3: Screenshots of the second phase of the doctor's appointment service requesting the input of an appointment date, rendered by the user interface application for mobile devices

Finally, a list of all available appointment dates fulfilling the desired date and time of day is presented to the user, allowing him or her to choose and book one of them. Again, a service version used in a real AAL environment would need to check the date and time of day with a remote web server in order to retrieve the available appointment dates. Since the current

version of the doctor's appointment service is only used for testing, random values are used instead. A list of random dates is presented to the user through a single input interaction of type `SingleChoice` that is implemented by the user interface application for mobile devices as a list of radio buttons.

After one of these appointment dates has been selected, the user is asked for confirmation in order to finally reserve the chosen appointment. The service prototype realizes this through a group operation including the following two simultaneous interactions:

- The name of the selected doctor, including the exact street address, is presented in addition to the day and time of the selected appointment, using an output interaction of type `Text`.
- An input interaction of type `Command` allows the user to reject the selected appointment. If triggered by the user, the second phase is started over by asking the user again for selecting a different date and time of day.

The `Command` interaction mentioned above is implemented by the user interaction application for mobile devices as command button the user can tap on to indicate that the selected appointment shall not be booked but modified.

Figure 5.4: Screenshot of the doctor's appointment service requesting the user to confirm the selected appointment, rendered by the user interface application for mobile devices

The integration of this `Command` interaction is necessary since the use of the hardware back button - which would normally be used to navigate back a few steps and modify the appoint-

ment details - is not permitted by the user interface generation system, as explained in section 5.1. Figure 5.4 shows a screenshot of this confirmation, presented using the user interaction application for mobile devices.

The last step of the doctor's appointment service presents a confirmation to the user, including the selected doctor's name and address and the exact appointment date and time, as single output interaction.

5.2.2 Medication reminder service

The medication reminder service starts with entering reminder details. Since no guidelines for presenting list data could be established from the results of the first user workshops, the medication reminder service simply follows the approach of presenting all list items successively, each with the options to modify or delete the current item, followed by an empty screen for adding an additional list item. Applied to the list of reminder messages that is to be created by the user as first step of the medication reminder service, this means that several group operations are sent to the user interface application successively, each representing one reminder item.

For each reminder item, the exact reminder message must be defined, and the time of day when to activate the reminder. Each of these group operations representing one reminder item initiates three simultaneous input interactions, represented by the following three operations included in the group operations:

- The user is requested to input the desired reminder message through an input interaction of type `Text`.
- The time of day is to be entered through an input interaction of type `Time`. The accuracy value of the `Time` data type used is set in order to ensure that only hours and minutes are requested, and minutes may only be entered in intervals of quarters of an hour (0, 15, 30, 45).
- An additional input interaction of type `Command` is included to allow the user to indicate that no additional reminder items shall be created. This interaction is included in all successive group interactions except for the first one, since the user must specify at least one reminder item.

Such a group interaction is sent to the user interface application, and the results - reminder text and time of day - are stored internally by the service application in a list, until the user triggers the `Command` interaction to express that no additional reminders shall be added. In this case, the medication reminder service continues with the next phase of requesting a telephone number which is explained in the next paragraph.

Figure 5.5 shows two screenshots of the user interface application for mobile devices interpreting this first step of the medication reminder service: The first screenshot shows a request for the first reminder item to be specified, the required fields are already filled in and the option of skipping this step and not adding any reminder is not shown. The second screenshot shows

how the input of additional reminder items looks like after the first one has already been defined, this time the command button for finishing adding new items and switching to the next phase is displayed.

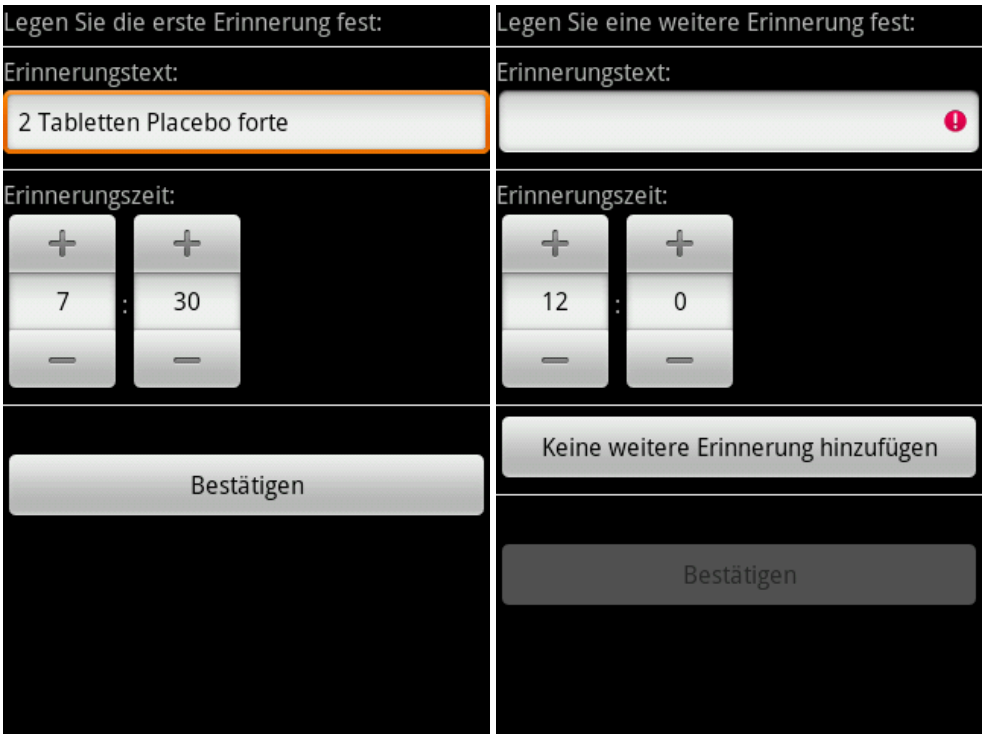


Figure 5.5: Screenshot of the medication reminder service requesting the user to specify reminder details, rendered by the user interface application for mobile devices. Both versions are shown, without (left) and with (right) the option of not adding additional reminder items.

In the second phase, the medication reminder service requires the specification of a contact person to be informed in case of emergency. This is done through a single input interaction of type `PhoneNumber`. As formal restriction, the minimum size of the phone number to be entered is set to three characters. In addition, the semantic flag `IS_KNOWN_BY_USER` is set to indicate that a contact that is already stored on the UI device may be chosen.

Figure 5.6 shows a screenshot of this phase, presented by the user interface application running on a mobile device featuring a contacts application. The input interaction is realized as text box in which the requested phone number can be entered, using an on-screen keyboard showing only digits and certain delimiting characters. Due to the `IS_KNOWN_BY_USER` flag being set and the UI device being able to show a contacts application, the option of choosing a phone number from this contacts application is provided through a separate command button.

After both the reminder messages and the emergency contact have been specified by the user, a confirmation message is shown through a single output interaction of type `Text`. This informs the user of how many reminder items have been stored by the service. When using

Figure 5.6: Screenshot of the medication reminder service requesting the input of an emergency telephone number, rendered by the user interface application for mobile devices

the user interface application for mobile devices, the content of this `Text` data type is simply displayed on the screen in textual form.

To be able to test also the activation of a reminder and a user's reaction to it, the medication reminder service has been implemented to wait for approximately 20 seconds after the confirmation message mentioned above has been displayed, and after this timespan the first medication reminder message is started independent of the time of day originally specified by the user in the first phase of the service execution.

The reminder itself is implemented as an `AlertOutputOperation` that contains a data item of type `Text`. This data item contains the exact reminder text specified by the user in the first phase of the service execution. When the user interface application for mobile devices receives this `AlertOutputOperation`, the reminder text is displayed on the screen as it is the case also with normal operations of type `OutputOperation`. In addition, however, a confirmation button is displayed, similar to the one used with input operations and group operations. Finally, also a sound message is played and the UI device starts vibrating. Both the sound and the vibration do not end until the user has confirmed taking the medication by tapping on the confirmation button. Figure 5.7 shows a screenshot of an active medication reminder, using the user interface application for mobile devices.

Figure 5.7: Screenshot of an active medication reminder, rendered by the user interface application for mobile devices

5.3 User tests

A second set of user workshops was performed to test the prototype of the user interface generation system in typical use-cases. The selection of the target group for these practical tests was similar to the first set of user workshops, the test persons were defined to be at least 55 years old and not suffering from cognitive defects such as dementia or Alzheimer's disease.

The user tests were performed in September 2012. A total number of seven test persons participated in the workshops, including three male and four female participants. They were between 55 and 86 years old. Three of those seven persons had already participated in the first set of user workshops. All workshops took place in the test persons' homes, and were performed individually or in groups of two persons.

The aim of the research project and the test procedure of the workshops were explained in detail to all potential test persons. Prior to starting the user tests, they were asked if they were willing to participate. They were informed that participation would be on a voluntary basis, and asked for their compliance to video- and audio record of the workshops.

The workshops consisted of three parts:

- As in the first set of user workshops, an interview was performed as the first step. Both the questions asked and the basic purpose of the interview were exactly the same as in the first workshops, as listed in section 3.5.1.

  The interview was only performed with those test persons that did not participate in the first workshop.

- In a second step, the test users were asked to test the two example services in realistic scenarios using a real mobile device. The assistance provided by the test supervisor was reduced to a minimum, and these practical test cases were video-recorded.

- In a final discussion, the test persons were asked to express their opinion of the system they were able to try in the second step as well as report problems they had in the course of the practical tests. In addition, they could suggest ideas for improvement of the overall system.

The overall purpose of the second set of user workshops was to test the prototype version of the user interface generation system, figure out whether potential users evaluate it as being qualified for practical use in AAL environments, and define which extensions and enhancements are most important to be implemented in future versions. In detail, the following questions should be answered:

1. Are the user interfaces generated by the system intuitively usable by the prospective user group?

2. At which points of the execution of the user interface application for mobile devices do problems in the interaction between user and system occur?

3. Which additional features are desired by the system's prospective user group?
   This concerns especially the following topics:

   - Which additional user interface devices are defined as most important by potential users for controlling AAL service, as alternative or in addition to mobile devices?
   - Is an approach that allows users to switch from one user interface device to another while executing an AAL service desirable for potential users?

Since only the second and the first steps of the user workshops differed from the first workshops performed prior to the prototype implementation, the exact test procedure and the purpose of those two steps - the practical tests and the final discussion - are discussed in this section. For details concerning the interview conducted as first step, see section 3.5.1.

To perform the practical tests, two exemplary scenarios were defined, one that tests interaction with the doctor's appointment service, the other one testing the interaction with the medication reminder service. The first scenario concerns the definition of two medication reminders, including the specification of the exact reminder text and time of day, followed by the selection of an emergency contact. This scenario also includes the activation of one medication reminder message and tests the users' reaction to the reminder alert. The second scenario resembles the task of searching for a certain doctor and booking an appointment with this doctor's office.

The practical user tests were performed using the same Samsung GT-I5500 smartphone that was also used in the first set of workshops. Again, both screen brightness and audio volume were set to the maximum value. The phone's settings were changed in order to ensure that a clicking sound would be played immediately after each tap on the touch screen, following the request for instant feedback suggested by one of the guidelines established as result of the first set of workshops, as documented in section 3.6.3.

To simulate the communication between the service and front end hardware layers, the service applications were executed on a laptop computer for the practical tests, while the user interface application was running on the above-mentioned mobile phone. The mobile phone was setup in order to act as WiFi access point, and the laptop computer joined the wireless network provided by the phone. This way, the two devices could communicate and exchange `Operation` and `Result` objects through socket connections.

Since some problems concerning the handling of touch screens occurred in the course of the first user workshops, special focus was on the instruction of the test persons prior to the second set of user tests. Those test persons that did not participate in the first workshops were presented several screens containing typical UI widgets which are often used by Android applications in order to get familiar with the look-and-feel of the Android operating system. They were also allowed to play around with these screens on the test device to practice the interaction with a touch screen interface.

All test users, also those who had already participated in the first set of workshops, were especially trained to the use of the hardware back button, since the handling of this button when working with the user interface application prototype differs from its typical use on Android devices. The test users were explained that the hardware back button could not be used to navigate back to a screen previously displayed, and that it could be used instead to hide the on-screen keyboard which might be useful if UI widgets are hidden behind the keyboard.

Prior to the practical tests, the test persons were presented two print-outs of the test procedure for each test scenario. These contained a general description of the purpose of each of the two example services, explaining which tasks may be accomplished through these services. In addition, they contained detailed instructions the test persons should follow during the practical tests.

The instructions given to the test users were formulated in order to not tell the users in detail which position on the touch screen to tap on at which point of time. Instead, they rather focused on the concepts behind each single user interaction, telling test persons which sub-tasks should be accomplished in which order. Nevertheless, the test instructions explained all necessary steps in detail, including all desired inputs and data selections. This was necessary because they should be detailed enough to allow test users to follow them on their own, without requiring constant instructions from the test supervisor. The exact wording of the test instructions in German language is shown in figures 5.8 and 5.9.

The test persons were asked to perform each of the two test scenarios on their own. The test supervisor was constantly watching, but assistance was offered only if a test user could not accomplish a sub-task after trying for a longer period of time. The reason for this test setup was that one of the aims of the practical tests was to evaluate whether the user interfaces automatically generated by the system were intuitively understood by the test users and could be used without assistance.

During the practical tests, the users' interactions with the test device were video-recorded to be able to reconstruct the exact interactions with the touch screen later on. Another reason for the video recording was that the records were used after the tests to evaluate how long it took each test user to accomplish the various sub-tasks of the two example scenarios.

After both practical tests were finished, the test persons' opinion of the system was discussed in an open conversation that was not guided by fixed interview questions. Topics covered by this discussion included problems test users had that were observed during the practical tests, as well as additional feature requests suggested by the users. In the course of this discussion, test persons were also asked if an approach that allows users to switch from one user interface device to another while executing an AAL service would be desirable and if they would make use of such an option. Similar to the interview, the results of this conversation were written down by the test supervisor.

Arzttermin

Der Dienst *Arzttermin* erlaubt einerseits, Fachärzte in Ihrer Nähe zu finden, und andererseits direkt Termine mit einem dieser Ärzte zu vereinbaren.

Um einen bestimmten Facharzt zu suchen, müssen Sie die gewünschte Fachrichtung angeben, sowie die gewünschte Krankenkasse mit der der Arzt einen Vertrag abgeschlossen hat. Die gefundenen Ärzte werden nach Entfernung sortiert und mit Adresse angezeigt.

Um danach gleich einen Termin zu vereinbaren, geben Sie den gewünschten Tag an und ob Sie eher vormittags, nachmittags oder den ganzen Tag über Zeit haben. Aus allen noch freien Terminen können Sie dann einen auswählen.

Aufgabe:

Starten Sie den Dienst *Arzttermin-Service*. Sie sollen nun nach einem Arzt suchen und einen Termin mit diesem Arzt vereinbaren.

Sie suchen nach einem Arzt für Haut- und Geschlechtskrankheiten. Sie sind bei der Krankenkasse BVA versichert, der Arzt sollte daher einen Vertrag mit dieser Kasse haben.

Vereinbaren Sie einen Termin für heute in einer Woche. Da Sie an diesem Tag bereits eine Verabredung am frühen Nachmittag eingeplant haben, soll der Arzttermin am Vormittag liegen.

Figure 5.8: Instruction sheet given to the test persons prior to the practical test scenario for evaluating the doctor's appointment service

Medikamentenerinnerung

Der Dienst *Medikamentenerinnerung* erinnert Sie zu bestimmen, vorher festgelegten Tageszeiten an die Einnahme Ihrer wichtigsten Medikamente. Wenn Sie innerhalb einer bestimmten Zeitspanne nicht auf die Erinnerung reagieren und die Einnahme des jeweiligen Medikaments bestätigen, wird telefonisch ein Angehöriger oder eine Pflegekraft verständigt.

Der Programmablauf besteht aus drei Schritten: Im ersten Schritt haben Sie die Möglichkeit, beliebig viele Erinnerungen zu erstellen. Für jede Erinnerung ist es notwendig, einen Text (etwa, welche Menge von welchem Medikament eingenommen werden soll) anzugeben, sowie die Uhrzeit zu der täglich die Erinnerung gestartet werden soll. Im zweiten Schritt geben Sie die Telefonnummer einer Person ein, die im Notfall verständigt werden soll. Im dritten Schritt werden die Eingaben bestätigt und alle Erinnerungen werden gespeichert.

Aufgabe:

Starten Sie den Dienst *Medikamentenerinnerung*.

Erstellen Sie zwei Medikamentenerinnerungen: Sie möchten jeden Tag um 9.00 Uhr daran erinnert werden, das Medikament „Orthomol“ einzunehmen, sowie um 15.30 Uhr an die Einnahme von 2 Tabletten „Placebo forte“.

Geben Sie an, dass im Notfall Frau Karin Wagner verständigt werden soll, sie hat die Telefonnummer 0664/6457544. Sie können die Telefonnummer manuell eingeben, oder Frau Karin Wagner im Telefonbuch auswählen wo die Nummer bereits gespeichert ist.

Nachdem Sie alle Eingaben bestätigt haben, warten Sie etwa eine halbe Minute bis die erste Erinnerung startet. Dann bestätigen Sie die Erinnerung, so als hätten Sie das betreffende Medikament eingenommen.

Figure 5.9: Instruction sheet given to the test persons prior to the practical test scenario for evaluating the medication service

5.4 Results

This section summarizes the outcomes of the second set of user workshops, including both the answers given in the course of the interviews and the observations made during the practical tests and the final discussion.

5.4.1 Interviews

Similar to the group of test persons participating in the first set of user workshops, the test users performing in the second workshops formed a rather homogeneous group regarding their habits of mobile phone use. All the test persons stated to own and regularly use a mobile phone, and they all have had their mobile phones for several years. Some details nevertheless were answered quite differently from the first interview procedure.

The following overview summarizes the answers to all interview questions:

- *Do you own a mobile phone?*

  All of the test persons stated to own and regularly use a mobile phone.

- *Which type of mobile phone is it?*

  Most of the test persons used a classic phone with keyboard. One person said to use an old smartphone, containing a large screen that does not feature touch interaction and a full hardware keyboard running Windows Mobile.

  One person stated to own and regularly use an Android-based smartphone featuring a touch screen. The exact model is very similar to the one used in the user tests, therefore the comments and results of this test person are especially interesting since she is the only person participating in the workshop that is experienced with and accustomed to the handling of an Android-based device.

- *When did you get your first mobile phone, and for which primary reason?*

  All test persons said that they have had a mobile phone for at least a few years, some even for more than 15 years. The main reason for acquiring a mobile phone was that it makes it easier to stay in contact with friends and relatives, especially with their children and grandchildren. Three of the seven persons stated that they got a mobile phone in order to replace an existing landline telephone.

- *What do you typically use the phone for, apart from simple phone calls, and how often?*

  Two of the test persons, among them the oldest participant, said they would use their mobile phone only for starting and receiving phone calls, one of them instead of a landline phone and therefore very often. All the persons participating in the interview stated to use their mobile phone at least a few times per day. Apart from starting and receiving phone calls, text messaging was mentioned as the most important use case.

In addition, the phone's camera, the contacts and calendar applications, and the possibility of composing and reading e-mails were mentioned. Three persons noted that they use their phones as replacement for an alarm clock. One person stated to use the possibility of installing third-party applications, for example dictionaries. The test person that uses an Android-based smartphone was the only one that mentioned browsing the Internet as use-case.

- *If you are not in your home, do you usually carry the mobile phone with you?*

  All persons stated to always carry the mobile phone with them when being on the way.

- *If you are at home, is there a common place where the mobile phone is situated, or do you carry it with you?*

  Four persons said the phone was nearly always with them, two said they that they were frequently looking for their phone in different places all around their home, and in one case a fixed place at home was provided as answer.

- *Is the mobile phone always turned on? If not, in which situations and how often do you turn it off?*

  This question showed that all persons except for one have their phones turned on all the time. Only one persons stated to turn it off at night.

- *Apart from mobile phones, which other electronic devices do you use?*

  All test persons except for the oldest one participating in the interview said that they would own and regularly use a personal computer, either a desktop computer or a laptop device. A television set was mentioned a few times, including supplemental equipment such as DVD player and hi-fi system. As additional devices, users mentioned radio and GPS-based car navigation system.

- *For each of these devices, how often and for which purpose do you use it?*

  Various use cases for personal computers were mentioned, including e-mail, surfing the Internet, writing and printing documents, displaying and enhancing digital photos, and recording and editing music. All persons owning a PC stated to use it frequently, some even daily.

None of the test persons mentioned an electronic device that he or she uses exceptionally more often than the mobile phone, apart from personal computers. This implies that for all test persons, the smartphone platform used in the user tests seems to be the ideal solution as AAL user interface device.

5.4.2 Practical tests and final discussion

Concerning the general handling of touch screen devices, similar observations as in the first set of user tests have been made during the practical tests. This especially applies to wipe gestures and the necessary amount of pressure for performing simple tap gestures. In the course of

the practical tests, persons not experienced in the use of touch screens tended to press command buttons displayed on the screen too hard and were confused if the device did not react because the touch screen did not recognize the tap.

In addition, wipe gestures - used for example to scroll through a list - were performed rather slowly and carefully, forming several short wipe gestures instead of a longer one. It seems that the test persons felt uncomfortable when staying with their fingers on the touch screen for a longer time, they quickly touched the screen, wiping in any direction, and then lifted the finger again.

Another observation that also concerns the general interaction with touch screens and that has not been observed during the first user workshops also regards list scrolling. It occurred especially to users unacquainted with the handling of touch screen devices but experienced in the interaction with personal computers through a mouse device. When performing wipe gestures for scrolling through a list, these persons moved the finger in the wrong direction. On most touch screen devices, to move down in a list the finger must be positioned anywhere on the list and moved upwards, following the metaphor of grabbing and moving the whole list object. On the other hand, when using a scroll bar widget in a graphical user interface with a mouse device, the scrollbar must be slid in the opposite direction. It seems that the test persons mentioned tried to apply this interaction method they are used to from working on a PC to the interaction with the touch screen.

While the command buttons and other UI widgets defined by the user interface application are in general displayed as large as possible, following one of the guidelines established as the result of the first set of user workshops, the prototype version of the user interface application still uses the standard on-screen keyboards offered by the Android system. Similar to the first user tests, observations showed that users not experienced with the handling of touch screen devices can hardly hit single buttons on the full on-screen keyboard, thus making it very hard for them to write full words as it is required, for example, by the setup phase of the medication reminder service.

Apart from those general problems that arise by a lack of experience with the interaction with touch screens, the following problems occurred in the course of the practical tests, that were either observed during the tests or mentioned by test persons in the course of the final discussion:

- One major problem is the fact that the hardware back button can not be used when interacting with the user interface application, due to technical restrictions as explained in section 5.1.2. The test person who has experience in the handling of devices running the Android operating system tried to use this button a few times and was confused since no reaction was provided by the device. During the final discussion, she pointed out that the presence of the back button is an important feature on Android devices and that users accustomed to working with the Android system might not understand why it does not react.

- Another problem that was faced by nearly all test users is that the confirmation button might be hidden if the other widgets require too much space on the screen. For example, the screen requesting the selection of a doctor consists of a list of radio buttons, followed

by the confirmation button. Since the list is longer than the available screen space, it has to be scrolled to reveal the confirmation button.

Most users expected that when selecting one doctor by tapping on the respective radio button, this selection is automatically stored by the system, and waited for the next screen to be displayed. The test supervisor had to explain that they needed to scroll to the end of the list, and tap on the confirmation button in addition to choosing a doctor's item.

- In the setup phase of the medication reminder service, a command button is shown that allows the user to stop the definition of reminder items and continue with the input of an emergency contact. The intention of this command button was not clear to all test users. Some entered the details of the first and the second reminder and then instantly tapped on this button, instead of first confirming the second reminder item using the confirmation button, and in the next screen triggering the above-mentioned command button to indicate that no additional reminder details shall be defined.

- One of the test persons who had already participated in the first set of user workshops apparently remembered the exact presentation of the *presentation of location-based data* interaction scenario (to review the original interaction scenarios used int the first user tests, review section 3.4.3). He noticed that in the doctor's appointment service, the presentation of the list of available doctor's offices is presented in a more simple way than in the *presentation of location-based data* interaction scenario in the first user tests.

  In fact, due to the limitations of the `SingleChoice` data type all information contained in each list item (each doctor's name, street address, and distance from the user's current position) is displayed using the same font size and color, while in the *presentation of location-based data* interaction scenario the doctor's name was emphasized by displaying it using a larger font size and a special text color.

  In addition, the same test user stated that he missed the map view showing the exact position of each doctor's office, since he liked it better than the textual list in the course of the first user workshops.

- Concerning the `NumberPicker` widgets used for the input of hours and minutes in interactions of type `Time`, and of day and year in interactions of type `Date`, not all test users were aware that these widgets allow two ways of entering data - the direct input through an on-screen keyboard, and the possibility of increasing and decreasing predefined values through + and - buttons. Some instantly used one of the two options and ignored the other one, however all test persons managed to enter the desired numbers.

  One test person noted that the use of the + and - buttons is not possible if no numeric value is present in the respective `NumberPicker` widget. This can be explained because there is no value that can be increased or decreased, but the widget could be designed more user-friendly if tapping on one of the buttons while the widget is empty would result in a default numeric value filled in.

In addition, one test person noted that for the interaction scenario of the medication reminder it would be desirable to create different medication reminders for different weekdays. However,

this is not related to problems or shortcomings of the framework and the user interface application but instead completely depends on the service application. For the practical user tests, the example services were intentionally designed as simple as possible.

5.4.3 Evaluated questions

Are the user interfaces generated by the system intuitively usable by the prospective user group?

Apart from the problems that were caused by lack of experience with the handling of touch screen devices, only short interruptions occurred due to a widget or screen being not self-explanatory to the test users. Examples for such user interfaces that caused problems are listed in the previous section.

Only details were criticized. This concerns, for example, the screen asking for the desired date of a doctor's appointment. When selecting certain months that consist of more than eight characters, the caption's value is too long to be displayed in the same row as the `NumberPicker` widgets for selecting day and year, and a line break is automatically included in the caption of the drop-down list for selecting the month. Therefore, the last characters of the month's caption may be displayed in a second row, looking weird to the test users.

Another example is the general font size used for the presentation of several simultaneous interactions. It is smaller than the font size used in single interactions, and two of the test persons criticized that text displayed using this smaller font size is difficult to read on such a small screen, especially when using black characters on colored background, as it is used in the content of many UI widgets such as the drop-down lists representing the `SingleChoice` and `MultipleChoice` data types.

In addition, one test user had problems in the interaction with the first screen of the medication reminder service that requests the input of reminder text and time of day. She did not know what to fill in the text box that was intended for the reminder text, since the caption defined in combination with the respective input operation of type `Text` was not self-explanatory. This is not a shortcoming of the framework or the user interface application since the caption is defined by the service provider, but it shows the importance of providing explanatory captions, especially if service providers have no influence on the exact positioning or presentation of UI widgets as it is the case in a user interface generation system.

At which points of the execution of the user interface application for mobile devices do problems in the interaction between user and system occur?

As mentioned above, most problems that occurred in the interaction were caused by misuse of the touch screen. Apart from that, some minor problems occurred that required the test supervisor's assistance, as listed in the previous section.

Which additional user interface devices are defined as most important by potential users for controlling AAL service, as alternative or in addition to mobile devices?

Most of the test persons generally approved the idea of executing assisting services using

different devices apart from personal computers. However, they had problems in the interaction with the example services mainly because of a lack of experience in working with touch screen devices. Due to these problems, most test users stated that they would rather use the mobile phones they are accustomed to - in most cases devices with a physical keyboard and a small screen that does not feature touch interaction - or laptop and desktop computers. Therefore these two platforms might be the most interesting ones for future implementations of user interface applications, followed by television sets since many users stated they would also own and regularly use a TV.

The test person who stated that he would miss the map view for choosing doctor's offices said he would also like to use an iPad or another tablet device, either for executing the whole doctor's appointment service or to switch from the smartphone to the tablet only for the selection of a doctor. The current version of the user interface application for mobile devices already supports tablet devices, therefore the execution of the two example services on a tablet device is not a problem. The integration of a map view for presenting the exact location of doctor's offices, however, would require adaptations to the predefined set of data types, for example an extension of the `SingleChoice` data type, as suggested in section 4.5.1.

In general, four of the seven test persons stated that a migratory approach that allows using several devices for the execution of a single AAL service would be an interesting option, but most of them added that they would only make use of such an option when working with more complex services that take a longer time to interact with. Only the one person that asked for a map view comparing the locations of doctor's offices, as mentioned above, said that he would switch from a smartphone to a tablet device to accomplish the single sub-task of selecting a doctor, mainly due to the larger screen available on tablet devices.

CHAPTER 6

Conclusions

6.1 Summary

The field of Ambient Assisted Living is of increasing importance in a society which faces the problem of an aging population. The use of AAL systems allows especially elderly people to live on their own in an environment they are accustomed to, assisting them in daily life, reducing the necessity of external care and assistance. This also opens an interesting and promising field for research work.

This thesis suggests the combination of the two research fields of Ambient Assisted Living and User Interaction Description Languages. The development of devices and creation of user interfaces for controlling and interacting with an AAL environment is a complex task. The application of abstract user interaction languages could facilitate this task, since user interfaces for existing devices could be generated in an automatic or semi-automatic way. Using such an approach would allow to control a highly complex system such as an AAL environment through consumer devices instead of being restricted to special devices or certain software for desktop computers.

First of all, the typical users of AAL systems would benefit from such an approach, since it would allow them to choose from a variety of devices for interacting with the AAL system, instead of learning how to handle a specialized device. Each user could choose the device he or she is accustomed to and is experienced with. This would also lower the acceptance threshold for AAL environments, because especially elderly people may be discouraged and refuse to use a system that requires handling an unknown technical device. In addition, the approach presented in this thesis is also beneficial for AAL service providers, since it allows the easy development of AAL services without the need of implementing user interfaces and constructing user interaction devices.

By establishing a list of seven criteria for evaluating the applicability of User Interaction Description Languages in AAL systems that especially take into account the needs of the typical user group, this thesis provides a basis for research work on the application of user interface generation in AAL environments. In addition, the detailed analysis of 15 existing User Interaction Description Languages and their comparison based on the newly established criteria shows that a variety of solutions following the proposed approach exist. However, the analyzed solutions differ a lot concerning their field of application. To form a complete user interface generation system for AAL environments, several of those solutions need to be combined since all operate using a different degree of abstraction.

Therefore, this thesis presents a user interface generation system that follows a slightly different approach. It includes both the definition of interactions between users and AAL services in an abstract and generic way, and the fully automatic generation of concrete user interfaces for different devices based on those interaction definitions.

Persons belonging to the typical target group of AAL services were integrated in the process of designing this system: Prior to implementing a prototype of the user interface generation system, the prospective user groups' typical handling of devices belonging to one class of potential user interface devices - smartphones - was examined. This investigation provided fundamental insights in the interaction of elderly people with smartphone devices. Guidelines for the design of the above-mentioned user interface generation system were derived from the results of this

investigation. The procedure and the results as well as the final guidelines are documented in detail, allowing their re-use and application to other research projects and software products.

This thesis furthermore documents the implementation of the first prototype version of the user interface generation system. The system's architecture, consisting of several components operating on different hardware layers, ensures that maximum accessibility is offered to users. At the same time, service providers benefit from the automatic generation of user interface implementations, saving time and effort compared to the manual creation of those interfaces for different devices. On the other hand, using such a framework that is based on strict rules and guidelines limits service developers' possibilities, as shown in the formal analysis of the system's prototype performed as part of this thesis in addition to the actual implementation.

The final evaluation, performed including test persons that belong to the system's prospective user group, shows that the fundamental idea of the approach suggested by this thesis works and is approved by potential users. Although it does not fulfill all criteria defined for evaluating a system's applicability to AAL environments, it proves that the automatic generation of user interfaces that provide users with maximum accessibility is possible. This guarantees that especially users suffering from physical disabilities are supported in everyday tasks in order to overcome their handicaps.

Both users and service providers need to accept compromises since the architectural and implementational details of the user interface generation system limit them in certain ways when compared to the use of traditional, manually created user interfaces. At the same time, however, it brings about various advantages that may compensate the above-mentioned shortcomings. For potential users of the system, the possibility of choosing from a variety of different user interface devices for controlling a complex system is one of the biggest advantages. Service providers might benefit mostly from the reduced effort necessary for providing user interfaces for many different types of devices.

The approach presented in this thesis might become a real alternative to the traditional concept of creating user interfaces for AAL service devices manually, but this is possible only if additional user interface applications for supplemental UI devices will be available. Most test persons participating in the two user workshops, for example, had problems in handling the example services using a smartphone because they had no experience with touch screen interaction. They would rather use classic mobile phones without touch screens, personal computers, or other devices using different modalities for interacting with AAL services.

6.2 Future work

This thesis documents the first iteration in the design and development of a software product, in this case a user interface generation system. The first four steps are described in detail, including analysis and comparison of existing solutions, user-centered design of the system's architecture, implementation of a first prototype version, and evaluation of this prototype through user tests including persons who belong to the prospective user group of the final software product. The purpose of this section is to suggest tasks to be performed as the next steps, with the aim of further developing the user interface generation system to a version that can be used in real AAL environments.

To schedule the detailed tasks for implementing future versions of the software product, additional evaluations and user tests will be necessary. First of all, practical user tests - similar to those documented in the chapter regarding the system's practical evaluations - should be performed, using tablet devices instead of smartphones. This is necessary since the prototype version of the user interface application aims at both smartphones and tablet devices, but the practical applicability to the hardware platform of tablet devices has not been tested in structured form.

In addition, large-scale user tests of the current version might be interesting, including additional AAL services, because only a small portion of typical AAL systems' possibilities are covered by the two exemplary AAL services defined and implemented as part of this thesis. Additional example services might also test and evaluate the communication with external sensors, thus opening a wide field of new opportunities. Prior to implementing new AAL services, it might be useful to adapt and extend the existing framework version, for example by adding supplemental data type specifications, as suggested in section 4.5.1.

An important step towards one of the main aims of the system - the possibility of choosing from a variety of different user interface devices - requires the implementation of additional user interface applications. In order to maximize the amount of different input and output modalities that can be used to control an AAL environment, this should include different types of devices. The interviews conducted in the course of the two user workshops show that, apart from mobile phones, personal computers and television sets are the types of devices mostly used by the typical user group of AAL systems. Especially the latter is a promising platform since it would allow users to execute AAL services using totally different input and output channels when compared to personal computers and mobile touch screen devices. In addition, a large-scale survey of elderly people could be useful to investigate which other consumer devices are desired to be integrated into the presented user interface generation system.

Some of the problems that occurred in the course of the practical user tests originated in the test persons' missing experience concerning the interaction with touch screens. Although most test persons stated that they would own and regularly use a mobile phone, only one of these phones proved to be a smartphone featuring a touch screen display. This implies that the hardware platform of mobile phones used as user interface devices is appropriate for the typical user group of AAL environments, but it does not apply to smartphones. Therefore a promising approach might be the adaptation of the existing user interface application to mobile phones which do not offer touch screen interaction.

Finally, also the idea of a migratory approach similar to the one suggested in section 4.5.3 should be reviewed, and this feature might be integrated in future versions of the user interface generation system. Some of the test persons participating in the user workshops for evaluating the prototype version of the system stated that they would make use such an extension that would allow them to switch from one UI device to another in the course of the execution of AAL services, however most of them found it useful only for more complex AAL services that the two that are part of the prototype implementation. Such additional AAL services must therefore be developed prior to realizing the migratory extension to the user interface generation system. Another prerequisite for such an extension is the integration of additional user interface device through developing user interface applications for different platforms.

Bibliography

[1] Wireless application protocol - wireless markup language specification. Technical report, Wireless Application Protocol Forum, 2000.

[2] The relationship of the UIML 3.0 spec. to other standards/working groups. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), 2003.

[3] Protocol to facilitate operation of information and electronic products through remote and alternative interfaces and intelligent agents. American National Standards Institute, ANSI/INCITS 389-2005, 2005.

[4] Information technology - user interfaces - universal remote console. International Organization for Standardization, ISO/IEC 24752, 2008.

[5] Information technology - user interfaces - universal remote console - presentation template. International Organization for Standardization, ISO/IEC 24752-3, 2008.

[6] Information technology - user interfaces - universal remote console - user interface socket description. International Organization for Standardization, ISO/IEC 24752-2, 2008.

[7] Adobe Developer Connection. *Developing applications in MXML*. `http://help.adobe.com/en_US/flex/using/WS2db454920e96a9e51e63e3d11c0bf69084-79b5.html`.

[8] Jan Alexandersson, Kai Richter, and Stephanie Becker. i2home: Benutzerzentrierte entwicklung einer offenen standardbasierten smart home plattform. In *Proceedings of USEWARE 2006 - Nutzergerechte Gestaltung technischer Systeme*, number 1946 in USEWARE'06. VDI-Verlag, 2006.

[9] Mir Farooq Ali and Marc Abrams. Simplifying construction of multi-platform user interfaces using UIML. In *Proceedings of the First European Conference on UIML*, 2001.

[10] Ali Arsanjani, David Chamberlain, Dan Gisolfi, Ravi Konuru, Julie Macnaught, Stephane Maes, Roland Merrick, David Mundel, T. V. Raman, Shankar Ramaswamy, Thomas Schaeck, Rich Thompson, Angel Diaz, John Lucassen, and Charles Wiecha. (WSXL) web service experience language version 2. Technical report, IBM, 2002.

[11] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language (second edition). W3C recommendation, W3C, 2010.

[12] Bert Bos, Tantek Çelik, Ian Hickson, and Håkon Wium Lie. Cascading style sheets level 2 revision 1 (CSS 2.1) specification. W3C recommendation, W3C, 2011.

[13] John M. Boyer. XForms 1.0 (third edition). First edition of a recommendation, W3C, 2007.

[14] Dick Bulterman. Synchronized multimedia integration language (SMIL 3.0). W3C recommendation, W3C, 2008.

[15] 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*, Vol. 15, No. 3, pages 289–308, 2003.

[16] David Chamberlain, Angel Díaz, Dan Gisolfi, Ravi B. Konuru, John M. Lucassen, Julie MacNaught, Stéphane H. Maes, Roland Merrick, David Mundel, T. V. Raman, Shankar Ramaswamy, Thomas Schaeck, Richard Thompson, and Charles Wiecha. WSXL: A web services language for integrating end-user experience. In *Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces*, CADUI'02, pages 35–50, 2002.

[17] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. W3C recommendation, W3C, 2007.

[18] James Clark. Comparison of SGML and XML. W3C note, W3C, 1997.

[19] James Clark and Steven DeRose. XML path language (XPath) version 1.0. W3C recommendation, W3C, 1999.

[20] Neil Deakin. *XUL Tutorial*. Mozilla Developer Network (MDN). `https://developer.mozilla.org/en/XUL_Tutorial`.

[21] Josefina Guerrero-Garcia, Juan Manuel Gonzalez-Calleros, Jean Vanderdonckt, and Jaime Muoz-Arteaga. A theoretical survey of user interface description languages: Preliminary results. In *Web Congress, 2009. LA-WEB '09. Latin American*, pages 36–43, 2009.

[22] James Helms, Robbie Schaefer, Kris Luyten, Jean Vanderdonckt, Jo Vermeulen, and Marc Abrams. User interface markup language (UIML) version 4.0. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), 2009.

[23] Kouichi Katsurada, Yusaku Nakamura, Hirobumi Yamada, and Tsuneo Nitta. XISL: a language for describing multimodal interaction scenarios. In *Proceedings of the 5th international conference on Multimodal interfaces*, ICMI'03, pages 281–284. ACM, 2003.

[24] Clayton Lewis. Using the "thinking aloud" method in cognitive interface design. Technical report, IBM T.J. Watson Research Center, 1982.

[25] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. USIXML: A user interface description language for context-sensitive user interfaces. In *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces with XML: Advances on User Interface Description Languages"*, AVI'04, pages 55–62, 2004.

[26] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. USIXML: a language supporting multi-path development of user interfaces. In *Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with the 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems*, volume 3425 of *EHCI-DSVIS'2004*, pages 200–220. Springer Verlag, 2004.

[27] Ashok Malhotra and Paul V. Biron. XML schema part 2: Datatypes second edition. W3C recommendation, W3C, 2004.

[28] Nigel McFarlane. *Create Web applets with Mozilla and XML*, 2003. `http://www.ibm.com/developerworks/web/library/wa-appmozx/`.

[29] Benjamin Michotte and Jean Vanderdonckt. GrafiXML, a multi-target user interface builder based on UsiXML. In *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, ICAS'08, pages 15–22. IEEE Computer Society, 2008.

[30] Microsoft. *XAML in WPF - .NET Framework 4*. `http://msdn.microsoft.com/en-us/library/ms747122.aspx`.

[31] Manuel Naujoks. XAML - WPF per XML. Hochschule Karlsruhe Technik & Wirtschaft, 2007.

[32] Matt Oshry, RJ Auburn, Paolo Baggia, Michael Bodell, David Burke, Daniel C. Burnett, Emily Candell, Jerry Carter, Scott McGlashan, Alex Lee, Brad Porter, and Ken Rehor. Voice extensible markup language (VoiceXML) version 2.1. W3C recommendation, W3C, 2007.

[33] Matt Oshry, Michael Bodell, Paolo Baggia, Daniel C. Burnett, Alex Lee, David Burke, Jerry Carter, Emily Candell, R. J. Auburn, Brad Porter, Ken Rehor, and Scott McGlashan. Voice extensible markup language (VoiceXML) 2.1. W3C recommendation, W3C, 2007.

[34] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction*, Vol. 16, No. 4, Article 19, 2009.

[35] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Model-based design of multi-device interactive applications based on web services. In *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part I*, INTERACT'09, pages 892–905. Springer Verlag, 2009.

[36] Steven Pemberton. XHTML^{TM} 1.0 the extensible hypertext markup language (second edition). W3C recommendation, W3C, 2002.

[37] Constantinos Phanouriou. *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, Virginia Polytechnic Institute and State University, 2000.

[38] Helmut Prendinger and Mitsuru Ishizuka, editors. *Life-Like Characters - Tools, Affective Functions, and Applications*. Springer Verlag, 2004.

[39] Angel Puerta and Jacob Eisenstein. XIML: A universal language for user interfaces. Technical report, RedWhale Software, 2001.

[40] Dave Raggett. *Getting started with VoiceXML 2.0*. W3C, 2001. `http://www.w3.org/Voice/Guide/`.

[41] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. W3C recommendation, W3C, 1999.

[42] Robbie Schaefer, Steffen Bleul, and Wolfgang Mueller. Dialog modeling for multiple devices and multiple interaction modalities. In *Proceedings of the 5th international conference on Task models and diagrams for users interface design*, TAMODIA'06, pages 39–53. Springer Verlag, 2007.

[43] Henry S. Thompson, Murray Maloney, David Beech, and Noah Mendelsohn. XML schema part 1: Structures second edition. W3C recommendation, W3C, 2004.

[44] Gregg Vanderheiden, Gottfried Zimmermann, and Shari Trewin. Interface sockets, remote consoles, and natural language agents - a V2 URC standards whitepaper. Technical report, URC Consortium, 2005.

[45] Jo Vermeulen and Jan Meskens. How to survive multi-device user interface development with UIML. Talk given at the DSP Valley/IBBT seminar on Exploring GUI design for embedded systems in Ghent on June 9th, 2009, 2009.

[46] Gottfried Zimmermann and Gregg Vanderheiden. The universal control hub: an open platform for remote user interfaces in the digital home. In *Proceedings of the 12th international conference on Human-computer interaction: interaction platforms and techniques*, HCI'07, pages 1040–1049. Springer-Verlag, 2007.

[47] Gottfried Zimmermann, Gregg Vanderheiden, and Al Gilman. Universal remote console - prototyping for the alternate interface access standard. In *Proceedings of the User interfaces for all 7th international conference on Universal access: theoretical perspectives, practice, and experience*, ERCIM'02, pages 524–531. Springer-Verlag, 2003.