FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Automatische Generierung Barrierefreier Grafischer Benutzeroberflächen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jasmin Kathrin Thöner, BSc
Matrikelnummer 1125020

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Roman Popp
Univ.Ass. Dipl.-Ing. Thomas Rathfux

Wien, 27. Dezember 2018

Jasmin Kathrin Thöner   Hermann Kaindl

Automatically Generated Accessible Graphical User Interfaces

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Jasmin Kathrin Thöner, BSc
Registration Number 1125020

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Roman Popp
Univ.Ass. Dipl.-Ing. Thomas Rathfux

Vienna, 27th December, 2018

Jasmin Kathrin Thöner	Hermann Kaindl

Erklärung zur Verfassung der Arbeit

Jasmin Kathrin Thöner, BSc
Schwendergasse 21-23/1/14, 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.

Wien, 27. Dezember 2018

Jasmin Kathrin Thöner

Danksagung

Ich möchte mich zuerst bei meinen Arbeitskollegen, und besonders bei Anna Tjagvad Madsen, bedanken, die mich auf das Thema Barrierefreiheit brachten. Ohne euch hätte ich dieses Thema wohl nicht für meine Diplomarbeit gewählt und mir wären dadurch einige wertvolle Erkenntnisse entgangen.

Ich bin meinem Betreuer, Prof. Hermann Kaindl, und meinen Assistenz-Betreuern, Dr. Roman Popp und Dipl.-Ing. Thomas Rathfux, sehr dankbar für ihre wertvollen Denkanstöße und ihr umgehendes Feedback. Ich bedanke mich auch dafür, dass sie mir die Gelegenheit gaben, mit ihnen ein Paper schreiben zu dürfen.

Ganz besonders möchte ich auch meiner Mutter und Jakob Englisch danken, die mir immer mit ihrer Unterstützung zur Seite standen. Eure Ratschläge und eure Motivation hat mir in meiner Studienzeit sehr geholfen.

Nicht zuletzt danke ich auch Clemens Heller für die großartige Zeit, die wir während unseres Studiums hatten. Ich bin froh, dass wir uns schon so früh in unserer Studienzeit kennengelernt und die gesamte Zeit über gegenseitig unterstützt haben.

Acknowledgements

First, I want to thank my colleagues at work, especially Anna Tjagvad Madsen, for leading me into the topic of accessibility. Without them, I might not have picked this topic for my thesis, and would have missed some valuable insights.

I am very grateful to my advisor, Prof. Hermann Kaindl, as well as my assistant advisors, Dr. Roman Popp and Dipl.-Ing. Thomas Rathfux, who always gave me valuable thought-provoking impulses and immediate feedback. I was glad to also have the opportunity of writing a paper together with them.

I want to thank my mother and Jakob Englisch for always being loving and supportive. Your motivation and advice has helped me a lot during my studies.

Last but not least I want to thank Clemens Heller for the great times we had during our studies. I am glad that we got to know each other very early on in our studies and for supporting each other all this time.

Kurzfassung

Die Entwicklung von Grafischen User Interfaces (GUIs) ist zeitaufwendig und fehleranfällig. Sie automatisch zu generieren senkt den Aufwand und lenkt gleichzeitig den Fokus auf die Geschäftslogik. Allerdings, da Frameworks zur automatischen Generierung von GUIs generisch sein müssen, um GUIs für die verschiedensten Anwendungsgebiete zu generieren, stellt Barrierefreiheit ein bedeutsames Problem dar. Diese Diplomarbeit untersucht anhand der Unified Communication Platform (UCP), einem modellgetriebenen Framework zur automatischen Generierung von GUIs zur Design-Zeit, wie barrierefreie GUIs automatisch generiert werden können und welche Einschränkungen hierbei existieren.

Eine Fallstudie wurde durchgeführt, um existierende Probleme bezüglich Barrierefreiheit in UCP aufzudecken. Eine Proof-of-Concept Anwendung sollte demonstrieren, wie diese Probleme mittels Responsive Design, Anreicherung semantischer Informationen und dem Austausch von Widgets zur Laufzeit überwunden werden können. Die Anpassung der Modelle und der Module zur Generierung des "Final User Interface" in UCP zeigen die Machbarkeit dieses Ansatzes. Eine Evaluierung des Ergebnisses gegen die Web Content Accessibility Guidelines 2.1 zeigt, dass UCP die meisten gesetzlich auferlegten Barrierefreiheit-Richtlinien erfüllt. Um manche Kriterien erfüllen zu können, muss jedoch der Model Designer über entsprechendes Wissen über Barrierefreiheit verfügen.

Diese Diplomarbeit zeigt, dass Frameworks zur automatischen Generierung von GUIs wie UCP gewisse Barrierefreiheit-Probleme lösen kann. Der Vorteil solcher Frameworks ist, dass dies immer reproduzierbar ist, da viele menschliche Fehler verhindert werden können. Die Anpassung des GUIs an verschiedene Endgeräte, besonders in Kombination mit Responsive Design, kann die Barrierefreiheit durch Optimierung des GUIs an die jeweils verfügbare Bildschirmgröße verbessern. Dasselbe gilt für die Ersetzung von Widgets zur Laufzeit, obwohl dies die Komplexität der Generierung erhöht. Der modellgetriebene Ansatz bewirkt, dass der Model Designer das resultierende GUI mit zusätzlichen Informationen, wie beispielsweise Texte für Screenreader, anreichern kann. Allerdings muss der Model Designer dafür entsprechendes Hintergrundwissen bezüglich Barrierefreiheit besitzen. Außerdem müssen Frameworks zur automatischen Generierung von GUIs laufend an technische Fortschritte und sich ändernde Barrierefreiheit-Richtlinien angepasst werden.

Abstract

The development of Graphical User Interfaces (GUIs) is a time-consuming and error-prone task. Automatically generating them the decreases effort while turning the focus on business logic. However, since automated GUI generation frameworks need to be generic in order to create GUIs for a variety of domains, accessibility is major concern. This thesis elaborates on how accessible GUIs can be automatically generated, and which limitations exist, using the Unified Communication Platform (UCP), a design-time model-driven GUI generation framework, as an example.

A case study was performed discovering existing accessibility issues in UCP. A proof-of-concept application demonstrated how these issues can be overcome using responsive design, enrichment of semantic information, and widget replacement at run-time. The modification of models and Final User Interface generation modules in UCP show the feasibility of this approach. Evaluation of the outcome against the Web Content Accessibility Guidelines 2.1 shows that UCP can fulfill most of the legally required accessibility criteria. Some criteria require the model designer to pay attention to them, i.e., he or she needs to have at least some knowledge about accessibility.

This thesis shows that automated GUI generation frameworks like UCP can eliminate a number of accessibility issues. The advantage of such frameworks is that this is always reproducible, since many human errors are avoided. Device tailoring, especially when combined with responsive design, can enhance accessibility by optimizing the GUI to the available screen space. The same applies to widget replacement, while increasing the complexity of the generation process. The model-driven approach to automated GUI generation enables the model designer to enrich the resulting GUI with additional information like texts for screen readers. However, the model designer needs to be educated about accessibility to provide necessary information to the models. Moreover, automated GUI generation frameworks need to be adapted to technical advances and updates of accessibility guidelines and regulations.

Contents

CHAPTER 1

Introduction

Graphical User Interface (GUI) development usually is a time-consuming, error-prone and, therefore, expensive task. Depending on the complexity of the user interface and the technologies being used, GUI-relevant code adds up to a significant amount of the overall source code. In 1992, Myers and Rosson [MR92] found that, on average, almost half of an application's source code is GUI-specific, while the time spending on user interface development makes up for 45% of the design phase, 50% of the implementation phase, and 37% of the maintenance phase [MR92]. In the last 25 years, a multitude of new user interface techniques (e.g., Web 2.0, Rich Internet Applications) and devices (e.g., smartphones, smart TVs, virtual reality headsets) have evolved, resulting in new and even more requirements than in the early 1990s [Pet07]. Therefore, the amount of time and code spent on an application's GUI has likely even increased.

Tools supporting automated generation of GUIs can decrease the software development effort by relieving developers from writing GUI-specific code. As a result, they can focus more on the business logic of their application. One approach to generate GUIs is to use model-driven engineering [Ken02], where the developer defines high-level abstract models of the tasks and domains involved in the user interface. These models are then automatically transformed into GUI-specific source code. This often involves various transformation steps, in which the models are iteratively refined and enhanced, e.g., to tailor the GUI to various ouput devices or fine-tune it through manual adaptations by the designer.

1.1 Motivation and Problem Statement

While model-driven automated GUI generation can help reducing the GUI development effort, it also faces various difficulties, especially regarding usability. Usability is typically lower when the GUI is automatically generated than when it is manually developed by experienced developers. One reason for this might be that GUI generation tools need to be generic in order to be applicable in various contexts, while usability also depends on the specific context of an application. Additionally, device tailoring poses a problem, since nowadays applications often are required to run on multiple devices (e.g., desktop computers, tablets, and smartphones). Both of these concerns have gained high interest in research (e.g., [AaIV08], [AVCF$^+$10], [CVC08], [FPR$^+$07], [KRF$^+$09], [RPK$^+$11]) and are continuously being addressed.

On the other hand, another important aspect to consider, namely accessibility, has not yet been addressed to a satisfying extent. Accessibility in the context of software is the ability of an application to be usable by people with physical or cognitive disabilities [OAS07] without limitations or help from others [HW16] [Wir]. Certain environmental conditions (e.g., noise, bright light, small screens, low bandwidth) can also influence the accessibility of an application [VJP11].

In Austria, and also the European Union in general, there are various laws that require Websites and software applications to be accessible. However, the government does not provide developers with detailed techniques or practices to achieve accessibility in their applications. Thus, one has to look for standards and guidelines that seem appropriate for the specific use case. Accessibility is a complex topic that depends on the content's type and size, as well as the complexity of the user interface and the used technologies. This context dependence makes it hard to ensure accessibility in automatically generated GUIs. Device tailoring further complicates this issue since the various device types have different needs and requirements regarding accessibility.

1.2 Approach

This thesis elaborates on how accessible GUIs can be automatically generated, using the Unified Communication Platform [PKR12] as an example. It also shows which possible limitations exist with regard to the models used for generation or the various types of output devices.

The Unified Communication Platform (UCP), developed by the Institute of Computer Technology at TU Wien, leverages a model-driven approach to generate GUIs at design-time from high-level *Discourse-based Communication Models* [FKH$^+$06]. While the framework already supports device-optimized GUI generation [RKP15b] [PRK13], accessibility has not yet been addressed, and hence the resulting interfaces currently are not fully accessible.

In a first step, existing legal regulations, standards, and guidelines (in Austria, the European Union, and worldwide) as well as the current state-of-the-art techniques for achieving accessibility in software applications were identified. Since the UCP framework makes use of HTML [W3Cc] as well as Apache Velocity [ASF10] and Eclipse JET [Ecl07], the literature study focused on techniques for achieving accessibility using these technologies.

Based on this background and related work, a case study was performed to discover the current limitations of UCP with regard to accessibility. Moreover, a set of requirements for meeting the legal demands and standards regarding the structure (and possibly behavior) of the GUI as well as the kind of output device was defined.

In a further step, a conceptual prototype for the adaptation and extension of UCP to fulfill these requirements was elaborated. An implementation of accessibility measures (based on the prototype) in the UCP framework was provided and evaluated using automated tools where possible, and using assistive technologies (e.g., screen readers) where necessary. Finally, challenges posed by the automatic generation of GUIs were determined.

1.3 Outline

Chapter 2 informs about the background of this thesis. The term “accessibility” is defined, legal aspects (i.e., laws and regulations relevant to Austria) of accessibility are discussed, and an overview of standards and guidelines is given. The technologies relevant for this thesis and their influence on acccessibility are discussed, and the process of accessibility evaluation is explained. A short introduction to automated GUI generation in general and the UCP framework in particular is given. Related work, focusing on different aspects of enhancing accessibility in automated GUI generation, is presented.

The case study that was performed to evaluate the current state of accessibility in UCP is discussed in Chapter 3. Six example applications have been evaluated using automated accessibility evaluation tools and assistive technology, and by manually checking their adherence to accessibility guidelines. Identified accessibility issues and lessons learned from the case study are discussed in detail.

In Chapter 4, accessibility improvements for UCP are proposed and implemented in a proof-of-concept application that demonstrates how the resulting GUI should look like when the improvements are implemented in UCP. The insights gained from a user study performed on the proof-of-concept application are presented. Finally, changes to the models needed to generate accessible GUIs are proposed and the adaptations of the module responsible for generating the GUI source code from these models are explained.

The adapted version of UCP is evaluated in Chapter 5. Chapter 6 discusses the results found during the evaluation and elaborates potential future work. The last chapter summarizes what has been done in this thesis and draws conclusions from it.

CHAPTER 2

Background and Related Work

This chapter informs about relevant background information and related work. Section 2.1 defines the term “accessibility”, explains its principles, discusses the various types of impairments an individual can be confronted with and points out the advantages of integrating accessibility into an application besides enabling inclusion of the disabled. Section 2.2 lists the various legal requirements posed by Austria, the European Union (EU), and the United Nations (UN). Section 2.3 discusses standards and guidelines which are currently available. Section 2.4 describes the influences technologies have on the accessibility of an application, focusing on technologies relevant to this thesis. Section 2.5 informs about ways of evaluating accessibility and which methodologies and tools can be used for an evaluation. Section 2.6 deals with the integration of accessibility engineering in the software development process. Section 2.7 gives an introduction into automated GUI generation using the model-driven development approach. Section 2.8 introduces the UCP framework and explains how the automated GUI generation process in UCP works. Section 2.9 reviews related work.

2.1 Defining Accessibility

Information and Communications Technology (ICT), and especially the Web, has become an integral part of our everyday lives. Nowadays, we can use it to communicate with others around the world, to buy products of any kind, to inform ourselves about the latest news, for educational purposes or simply for entertainment (e.g., audio or video streaming, playing games). Even more important, by now we are able to look for and apply to jobs or use governmental services (e.g., mobile signature, health insurance or financial services).

ICT has also enabled great opportunities for people with disabilities. Before the widespread availability of ICT, blind individuals barely had access to written text (e.g., newspapers, books), since they would need to buy expensive transcriptions to braille versions of the text or audio tapes. Alternatively, they would need someone else to read it to them, preventing them from self-determined access to information. With ICT, blind people can now use assistive technologies like braille displays or screen readers to access information whenever they want without relying on expensive transcriptions or other people. Likewise, individuals with motor disabilities, e.g., people bound to a wheelchair, can now use the Internet to do their shopping, therefore saving time and not having to rely on the physical accessibility of the shops.

Despite the fact that ICT opens up many chances for disabled people, it is often the case that developers do not pay enough attention to the *accessibility* of their applications, resulting in barriers that make it difficult or even impossible for disabled people to use them. After all, people with disabilities cannot use the applications in the same ways as people without disabilities can do. They need the information to be prepared in special ways (e.g., subtitles for videos so that deaf users can read what is said in the video), or they use various kinds of assistive technologies (see Section 2.4.4) which need to be able to process the information.

Access to ICT is a basic human right (see Section 2.2.3). Therefore, people with disabilities must be able to use ICT applications without particular limitations or help from others [HW16] [Wir]. Of course, they have special needs and ways to use those applications, but they have to be able to access all the information and perform all the actions (or equivalent ones) provided by them. In the broader sense, accessibility does not only apply to disabled people, but also to elderly people (who experience a gradual deterioration of their sensual capacities and cognitive abilities) and to certain situational or environmental conditions (e.g., small screens, noisy environments, bright light, low bandwidth) [VJP11].

Accessibility and *usability* are closely related. The Austrian government even includes usability in their definition of accessibility for software applications [RV16]. Usability deals with the ability of a user to achieve his or her goals in an effective, efficient and satisfying way using the application [ISOa]. Another closely related term is *universal design*, also called *design for all* or *inclusive design*, which defines a process to provide ICT applications for the biggest possible target group, involving everyone to the greatest extent possible [W3Cb]. Universal design also includes aspects like digital literacy, education, cultural aspects, or language.

2.1.1 Principles of Accessibility

The Web Content Accessibility Guidelines (WCAG) [CCGV08] define four principles of accessibility. The information presented to users in an application and its user interface components must be:

- *perceivable*, i.e., they must be available to their senses. For example, a deaf user cannot perceive the information provided by an audio track, if there is no written transcription available.

- *operable*, i.e., interactions with the application must be designed in such a way that all users can perform them. If a user interface is only operable using a mouse, people that rely on a keyboard and cannot use a mouse (e.g., people having tremors that prevent them from fine muscle control) would not be able to interact with it.

- *understandable*, i.e., information and interactions in an application must be understandable by the users. An icon button (e.g., a pencil icon to edit an input) without any kind of textual description cannot be understood by a blind user, since he cannot see the icon.

- *robust*, i.e., they must be able to be presented to a wide variety of user agents, including assistive technology, also with regard to their evolution (future versions should also support the application). If the application does not support screen readers, for example, blind users will have a hard time to access it.

These principles are expressed in verifiable success criteria, e.g., in the European standard EN 301 549 [EN315].

2.1.2 Types of Impairments

A person is disabled or impaired, if he or she cannot participate, or can only participate with difficulty in everyday activities, because of personal characteristics or external influences. The World Health Organization (WHO) defines the term “disability” as a situation in which a person cannot do something because of a barrier. This social model sees disability as a problem caused by society instead of as a problem of the individual. The problem could be removed if the barrier is removed, such that the person can perform the given task (even if it is done in a different way than the average person would do). In the following, the different kinds of disabilities are discussed. They might be congenital, or originated from an illness, a disease, an accident or developed with age. An individual may have more than one of these impairments. Disabilities can be temporary or permanent, and they may change over time (e.g., progressive or recurring impairments, situational or environmental differences).

Disabilities of Sight

Reduced vision is quite common today. Many people wear glasses or contact lenses to correct their eye-sight. However, sometimes one’s vision is so weak that these visual corrections are not enough to compensate. In Austria, this is the case for approximately 3,9% of the population [BMA13]. Very low vision often originates from certain diseases like macula degeneration, cataract, glaucoma, or diabetic retinopathy, but it can also result from aging. In order to be able to read text on a screen, people with low vision can use tools like screen magnifiers, which zoom in on a small part of the screen. Applications should also provide means of enlarging text and images. Additionally, high contrast between a text and its background can significantly facilitate reading on screens. Some people with low vision even use their own high contrast color scheme to overwrite the provided one. In general, applications can be made accessible for people with low vision by making everything (i.e., text, layout, as well as text and background colors) configurable.

An individual whose visual acuity is less than 2% is termed legally blind. Using computers with no eye-sight can be challenging, since neither a screen nor a mouse can be used. Hence, blind users use special equipment and tools which translate textual contents to tactile or audible information, and they heavily rely on the keyboard. Many use screen readers, which are software programs that output text in synthesized speech, so the user can listen to the content. Other tools for outputting text are refreshable braille displays, which use small pins to form braille characters. Typical devices can display 40 to 80 of such characters at the same time. As input devices, blind users mainly use a keyboard, sometimes they also use speech recognition software. Using ICT, blind users face issues when applications are not usable only by keyboard or when information is exclusively presented visually (e.g., if color-based cues are used or graphics containing important information are not accompanied by a textual description). Since screen readers do not just read from a screen from top to bottom, but are able to recognize the structure of the content, blind people can, for example, just browse through the headlines or links and quickly find the information they are looking for. This requires the content to be annotated accordingly (e.g., using correct tags in Hypertext Markup Language (HTML)). Issues arise if content is not structured properly or is presented in a confusing order, or when text is presented that is not meant to be read character by character (e.g., long URLs).

A third kind of sight impairment is a disability to perceive colors or brightness correctly. Color blindness is a condition that mainly affects men, and which can occur in various forms. The most frequent form is red-green deficiency, i.e., one cannot differentiate between red and green. In very rare cases, people can see no color at all, i.e., they only perceive shades of gray, since their eye's cones are not functioning. Color-blind individuals can have difficulties if information is presented solely by colors or if colors they cannot differentiate are too similar in saturation or brightness.

Auditory Impairments

Like visual impairments, auditory disabilities have a wide range in severeness, from mild declines in hearing abilities to deafness. 2,5% of people in Austria suffer from auditory impairments that cannot be corrected by hearing aids [BMA13].

People hard of hearing, i.e., with mild to moderate hearing loss, can have difficulties in understanding speech, especially when there is a lot of background noise, even when they use hearing aids. It can be helpful to provide subtitles or transcripts along with audio or video contents, and to provide high-quality audio content for which the volume can be adjusted.

When making ICT applications accessible to deaf people, one has to differentiate between congenital and acquired deafness. Individuals that are deaf since birth or acquired deafness before they begin to speak (prelingual deafness) have difficulties developing speech and language skills, since they have to rely on visual cues like lip reading. Thus, they may experience difficulties understanding complex texts. Most written languages use (combinations of) symbols that represent acoustic signals, and people that have learned to use speech imagine the sound of the formed words and sentences while reading to better understand their meaning. A congenitally or prelingually deaf person cannot use this technique, since he or she has never learned the sound of a language, but instead memorized which combinations of symbols represent certain concepts. This is similar to sign language, in which each sign has a specific meaning. As a result, individuals with congenital or prelingual deafness may have a harder time understanding difficult texts than people who acquired deafness after having learned to speak.

Providing translations for audio content into sign language can be a good way to enable deaf people to access the spoken content, but providing this as the only means for accessing the content is most likely not enough. Not all deaf people are able to use sign language, they may have just recently become deaf or they may have never learned a sign language at all. Additionally, there are many different sign languages. Even for the English language alone, there are various kinds of sign languages, e.g., American Sign Language, British Sign Language, or Australian Sign Language. These sign languages are not compatible, i.e., a deaf person using one of these languages does not understand the others. Thus, also textual transcriptions should be provided for audio content.

Motor Impairments

By far the most frequent permanent disabilities in Austria are motor impairments, making up for 13% of the population [BMA13]. Motor disabilities become an issue for ICT accessibility, if the fine motor skills of the arms and fingers are affected (e.g., by having tremors), or if a person cannot use typical input devices (i.e., mouse, keyboard, touchscreen) at all (e.g., because the person is paralyzed).

Individuals suffering from tremors or other kinds of conditions that impact their fine motor skills may have problems with clicking small areas or they need more time to interact with the GUI. They also might not be able to type simultaneous keystrokes. Hence, applications should provide long enough time frames to perform an action and provide alternative commands that do not involve simultaneous keystrokes. They also should be accessible by mouse or by keyboard only, and their clickable areas should be large enough. Error correction options as well as clear indication of current focus areas can also be of great help.

Some individuals with motor disabilities rely on alternative types of input devices, for example, if they are quadriplegic (i.e., partial or complete paralysis of both the arms and legs) or they have missing limbs. There is a wide variety of these devices, some of the most commonly used ones are:

- *Mouth stick / head pointer:* A mouth stick is placed in the mouth and used to tap on a touchscreen or press the buttons of a keyboard. It is a low cost tool and thus one of the most popular devices people with motor impairments use. With head pointers, the stick is mounted on the individual's head and the person can control it with head movements. Since using each of these devices quickly becomes tiring, interactions should be as easy as possible and not involve any unnecessary movements.

- *Oversized trackball mouse / joystick:* Trackballs or joysticks are easier to use for people with tremors, because you can scroll with the trackball or joystick, and then use separate buttons to perform a click. Sometimes, trackballs are even operated with a foot.

- *Adaptive keyboard:* There are various kinds of adapted keyboards available. In some cases, the space between each key is bigger than usual, or the areas between the keys are raised (instead of lowered), so that it is easier to place a finger over the right key. There are also overlays for standard keyboards. Additionally, word-completion software can be used to save some keystrokes.

- *Eye-tracking:* If an individual has a very limited range of motion or if movement is too tiresome, eye-tracking can be used. Using eye cameras, the movements of the eyes are tracked and projected onto the screen. Thus, the person can navigate through the application just with eye movements. Clicks can be simulated, for example, if the person focuses a clickable area with the eyes. Word-completion software can be used here as well, to save time when typing.

- *Voice recognition software:* If the speech producing organs are not affected by a disability, voice recognition software can also be used to interact with applications. If the software is trained properly, this is much quicker and less tiring than using any alternative input devices.

Cognitive, Speech, and Neurological Disabilities

There is a wide variety of cognitive disabilities that affect very different kinds of abilities, such as memory, problem-solving, reading / listening / verbal comprehension, attention, or learning. Some people with difficulties in comprehension of contents use, for example, screen readers to listen to the information while reading it at the same time. Others disable animations, and change text colors or spacing, to concentrate on the text. A clear structure of the content, good navigation, and easy-to-read texts aid in understanding the contents and purpose of an application. Predictable behavior of the GUI, different ways of navigation (e.g., breadcrumbs, search options, hierarchical structure of the navigation, site maps) as well as clear and consistent labels of input fields or interaction forms are also important. Some people with difficulties in problem solving can get easily frustrated, if error messages are unclear. The application should provide error prevention and correcting means, like suggesting alternate terms for search inputs or offering auto-completion of input terms. Distracting GUI elements such as pop-ups or animations should be avoided whenever possible.

People with speech disabilities experience difficulties producing speech, so that voice recognition software cannot recognize it properly. Hence, interaction with a GUI should not rely on voice alone.

In some people, certain animations like flickering, flashing or strobing effects (and also some optical illusions that suggest motion in a pattern) can trigger seizures. Thus, animations should not exceed certain thresholds for the frequency, contrast or size of these effects (e.g., WCAG provides such thresholds). In general, animations should be avoided if they are not necessary, because besides the potential risk of seizures in some people, they can also distract users or cause nausea.

Elderly People

Aging usually is accompanied by a gradual decrease in visual, auditory, memory, or motor skills. These effects can be mild, but they can also become disruptive. Often, there is a combination of impairments in multiple areas. In addition, inexperienced ICT users are more likely to be among the elderly. Nowadays, approximately 18% of the Austrian population are 64 years old or older [WPW12], and a shifting age distribution will increase this number in the future.

Environmental and Situational Influences

Besides the disabilities described above, there might also be temporary conditions that pose barriers for accessing ICT applications. These environmental and situational influences can be, among others:

- *Inexperience with ICT:* Inexperienced users are not yet familiar with interaction mechanisms of GUIs. Thus, GUI developers should adhere to common concepts and make interactions and representations as easily understandable as possible.

- *Low network bandwidth:* Applications relying on the Internet should also be accessible if the network bandwidth is low. An example would be to additionally provide a text-only version of the content for such situations.

- *Bright light:* Bright light can make text unreadable because of reflections on the screen. Providing a measure to increase the contrast of GUI elements can help in such cases.

- *Noisy environment:* Noisy environments pose difficulties for understanding audio contents. Thus, transcriptions or captions should be available, so that users can read the textual version of the contents in such a situation.

- *Mobile devices:* Mobile users can face a variety of the influences mentioned above: low network bandwidth (or temporary connection disruptions), bright surrounding light (e.g., the sun shining on reflective displays), a noisy environment, as well as a small screen. Because the screen usually is much smaller than a typical laptop screen or stand-alone display, much less content can be displayed on it at any point in time. Therefore, special techniques of GUI design are needed to provide the user with an intelligible interface and good usability.

2.1.3 Advantages of Accessibility

In addition to the inclusion of the user groups described above, there are various other benefits in implementing good accessibility in ICT applications:

- *Helps closing the "digital divide":* This term describes barriers posed for people without disabilities, e.g., of economic or social origin (low digital literacy, limited access to ICT, cultural differences, language barriers). An example would be to support a wide range of user agents, i.e. also less advanced and, therefore, cheaper ones. An application that is also usable with low network bandwidth enables people with no access to high-speed infrastructures to use them (e.g., in rural areas).

- *Mobile access:* Since accessible applications support small screens, keyboard-only interactions (i.e., to quickly change between form fields by activating the "enter" key instead of having to touch the next form field), high contrast interfaces, bigger fonts and interaction controls, as well as low bandwidth connections, they can be easily used on mobile devices.

- *Good basis for Search Engine Optimization (SEO):* Well-structured content, alternative texts for images and multimedia, enabling of keyboard-only interaction, and meaningful links make accessible Web applications easy to be indexed by search engines.

- *Larger target group:* Due to the benefits mentioned above, there is a much wider audience reach possible with accessible applications than with those which are not accessible. This also leads to other economic benefits like higher sales rates, or expanded market share.

- *Higher quality software:* Since accessible applications need to be future-proof in terms of advancing technologies and user agents, and accessibility specifications demand adherence to technical standards, accessible applications usually have higher quality.

- *Reduced maintenance costs:* The robustness of accessible applications and their adherence to technology standards improve their compatibility with future versions of user agents or technologies. Incorporating accessibility right from the beginning of development prevents from high changing costs, in terms of money and time, when the development is already finished.

- *Better reputation:* Social inclusion implied by good accessibility can increase customer loyalty. In fact, failing to provide accessible applications harms the image of the provider.
- *Adherence to policies and reduced legal risk:* The next section describes what laws there are regarding the accessibility of ICT applications. Providing accessibility also reduces the risk of lawsuits, which imply significant legal and reputational costs. Implementing the highest available standards also complies with more restrictive requirements introduced in the future.

2.2 Legal Aspects of Accessibility

Governments worldwide have defined various laws and regulations regarding accessibility in general and software accessibility in particular. In the following sections, regulations defined by Austria, the EU and the UN are discussed.

2.2.1 Austria

Austrian law regarding accessibility applies to people with disabilities. The Austrian government defines disabled individuals as individuals with any kind of non-temporary physical, mental, psychological, or sensory impairment, which potentially impedes participation within society (BGStG, BGBl. I Nr. 82/2005, Art 1 §3 and BBG, BGBl. Nr. 283/1990 §1 Sec. 2). A non-temporary impairment is one that lasts more than six months.

Bundesverfassungsgesetz (B-VG) - Article 7

The federal constitution explicitly states that individuals with disabilities must not be at a disadvantage. Equal treatment of people with and without disabilities is to be guaranteed in each area of everyday life. Any disadvantage for disabled individuals, whose removal is legally possible and reasonable, is defined as a discrimination and can be legally challenged.

Bundes-Behindertengleichstellungsgesetz (BGStG) - § 6 Section 5

Technical implementations of daily use and information processing systems (i.e., software) have to be accessible, that is, they can be accessed and used by disabled individuals normally, without limitations or help of others. This law was implemented on January 1, 2006.

E-Government-Gesetz (E-GovG) - § 1 Section 3

The e-government law states that governmental, administrative and business Websites must adhere to international standards for accessibility. The *General conditions of contract for IT services and software of the Austrian republic (AVB-IT)* require the contractor to provide an accessibility statement which states that the software conforms to the ÖNORM EN ISO 9241-171:2008 11 01 and at least level A of the WCAG (both discussed in Section 2.3).

2.2.2 European Union

In the European Commission's 10-year strategy *Europe 2020* for the economic advancement of the EU, started in 2010, one of the flagship initiatives is the *Digital Agenda for Europe*. The initiative's goals encompass accessibility of IT services, considering not only disabled people (as in Austrian law), but also elderly people, and people lacking digital literacy. The *eGovernment Action Plan 2016-2020* includes the initiative *"Inclusiveness and accessibility"*, which requires digital public services to be accessible for elderly and disabled individuals by 2020.

Directive of the European Parliament and of the Council on the Accessibility of the Websites and Mobile Applications of Public Sector Bodies

The most recent European directive on ICT accessibility was published on July 18, 2016 and addresses the accessibility of Websites and mobile applications of the public sector. The goal of this directive is to unitize the various laws and regulations at national level which were introduced due to the ratification of the UN Convention on the Rights of Persons with Disabilities (see 2.2.3). Until harmonized standards are published in the *Official Journal of the European Union*, the requirements defined in *EN 301 549* (see Section 2.3.4) are determined as the minimum requirements for Web and mobile accessibility. Public sector bodies should provide an accessibility statement claiming the compliance to the requirements posed by this Directive and a feedback mechanism should be implemented, so that people can report an application's shortcomings in accessibility. Member States are encouraged to provide awareness-raising and training programs for relevant stakeholders.

2.2.3 United Nations

Austria was one of the first EU member states that ratified the *UN Convention on the Rights of Persons with Disabilities* in 2008. The convention requires the states parties to *"take appropriate measures to ensure to persons with disabilities access, on an equal basis with others, [...] to information and communications, including information and communications technologies and systems, and to other facilities and services open or provided to the public, both in urban and in rural areas"*. This includes the implementation of minimum standards and guidelines in public digital services. These measures should be applied in an early development stage to minimize costs.

2.3 Accessibility Standards and Guidelines

In 1997, the World Wide Web Consortium (W3C) [W3Ca] called the Web Accessibility Initiative (WAI) [WAI] into life in order to develop standards, guidelines, and resources for creating accessible Web applications. The international standards proposed by WAI are the most referenced standards regarding Web accessibility, among them the WCAG, which form the basis of Austrian and European ICT accessibility laws. W3C standards are called W3C Recommendations and are developed in a structured process involving various steps: At the beginning, a Working Group publishes a *Working Draft* that is reviewed by the community and iteratively improved by the Working Group. The last version to be reviewed by the community is called the *Last Call Working Draft*, which includes the whole document for the future standard. This draft then becomes a *Candidate Recommendation*, i.e., an already quite stable guideline that can be implemented by developers. While implementing the guideline, developers report back their experiences and issues with it to the Working Group, which then adapts the Candidate Recommendation to further improve it for technical implementations. When the guideline is fully implemented, the Candidate Recommendation becomes a *Proposed Recommendation* and is ready for the final endorsement by the W3C. When approved by the W3C and there is sufficient support by the public, the Proposed Recommendation becomes a standard, which in case of W3C standards is called a *W3C Recommendation* or simply *Web Standard*.

Other international standards, dealing not only with Web applications, but with ICT applications in general, are provided by the International Organization for Standardization (ISO). ISO standards are developed by a technical committee consisting of experts in the industry and undergo a similar process as W3C Recommendations: A proposal for a new standard is proposed and, if accepted, a working group develops a working draft, which is iteratively improved. If all ISO members agree on the final draft (via voting), it is declared as an *ISO International Standard*.

Alongside accessibility laws, the EU and Austria also provide standards, especially EN 301 549 and ÖNORM EN ISO 9241-171, respectively. These standards reference industry standards like the WCAG.

2.3.1 Web Content Accessibility Guidelines

The WCAG 1.0 was published as a W3C Recommendation in May 1999. It has been revised and extended when in December 2008, the WCAG 2.0 was published. The most recent version of WCAG is 2.1, which became a W3C Recommendation in June 2018. Today, WCAG is a well-established standard that is widely used and also referenced by accessibility laws worldwide. It also became an ISO standard in October 2012 [ISOb].

It contains 13 *guidelines* which are organized into four *principles*: perceivable, operable, understandable, and robust. Each guideline expresses a basic goal and contains multiple testable and technology-independent *success criteria*. Guidelines and success criteria are explained in a separate document in detail, so that developers and evaluators can understand the issues disabled individuals can face with Web applications. Overall, there are 78 success criteria, which are organized into three *levels of conformance*: A (lowest), AA, and AAA (highest). A conformance level is reached, if all its success criteria, as well as all success criteria from lower levels, are satisfied. For each guideline, the WCAG Working Group also provides a set of *sufficient and advisory techniques*. Sufficient techniques correctly meet the success criteria, while advisory techniques typically improve accessibility, but might not fully meet the success criteria or are not (yet) supported by some of the current assistive technologies or user agents. There are also documented *failures* which are useful for identifying accessibility issues.

In September 2013, the WCAG Working Group published a note on the *"Guidance on Applying WCAG 2.0 to Non-Web Information and Communications Technologies (WCAG2ICT)"* [CKAV13]. This Working Group Note describes how the level A and AA success criteria can be applied to non-Web ICT applications.

Along with the WCAG, the WAI also provides two more sets of guidelines. The *Authoring Tool Accessibility Guidelines (ATAG)* 2.0, a W3C Recommendation since September 2015, describe how authoring tools (i.e., software used for creating Web and multimedia content) can be made accessible as well as how authors can produce accessible Web content using such tools. The ATAG requirements are also divided into the three levels A, AA, and AAA. The *User Agent Accessibility Guidelines (UAAG)*, on the other hand, addresses the accessibility of user agents (e.g., browsers or multimedia players). Currently, only UAAG 1.0 is a W3C Recommendation, but it is recommended by WAI to use the UAAG 2.0 Working Group Note from December 2015, since it is essentially complete. The only reason that it did not become a Candidate Recommendation was that sufficient testing resources were not available. It is not further developed by W3C, therefore it would not become a W3C Recommendation.

At the time being, a set of accessibility guidelines named "Silver", is currently in the works. While the WCAG 2.1 only focuses on the most urgent issues that were missing in WCAG 2.0, "Silver" will be a substantial revision of the WCAG, focusing on a broader scope (i.e., being less focused on Web technologies). WAI claims it will be easier to use and include more disabilities (e.g., pain). The project is currently in the research phase, and is said to be released in 2020.

2.3.2 Accessible Rich Internet Applications Suite (WAI-ARIA)

WAI-ARIA aims to make Rich Internet Applications (RIAs) accessible to assistive technologies (especially screen readers and keyboards). RIAs are dynamic Web applications with advanced user interface controls (e.g., expandable menus, drag-and-drop features) and dynamic content loading without requiring a page refresh. WAI-ARIA is an extension to HTML which enriches the semantic information a Web application and its widgets by providing a set of attributes which are typically preceded by the prefix "aria-". These attributes are interpreted by the user agents (e.g., browsers) and mapped to the operating system's accessibility API, which is used by assistive technologies to access the application. WAI-ARIA 1.1 is a W3C Recommendation since December 2017.

A key concept of WAI-ARIA is that of *roles*. The role attribute can be applied to an HTML element to describe its meaning. On the one hand, roles are used to describe user interface controls for which there are no equivalents in HTML. For example, there are various `div` elements that represent a tab panel with multiple tabs. Then, the outermost `div` element can be annotated with the role `tabpanel`, while the inner `div` elements are marked with the `tab` role. On the other hand, roles can be used to organize the document structure and as navigational landmarks. Using them can help making the intention of a certain area in an application clear (e.g., `toolbar`, `math`, `article`). Navigational landmarks, like the roles `navigation`, `main`, or `search` enable users to directly navigate to a certain region of the application (e.g., directly jump to the main part and skipping navigation).

Another important concept is that of *states* and *properties*. Properties can be used to, e.g., provide labels or keyboard shortcuts for HTML elements. States are used to describe the current state as well as state changes for elements, e.g., indicating that an element is disabled, invalid, or selected. There could be, for example, a slider bar, for which one can specify its minimum and maximum values as well as its currently selected value.

Live regions are the third key concept of WAI-ARIA and are used to indicate dynamic updates (e.g., updating of other user's states in a contact list of a chat application). This is especially useful for screen readers, so that blind users can perceive that the data has been updated.

2.3.3 ÖNORM EN ISO 9241-171

The *ÖNORM EN ISO 9241-171:2008 11 01 – Ergonomics of human-system interaction – Part 171: Guidance on software accessibility* provides guidelines for general ICT applications (i.e., not only for Web applications) and therefore can be seen as a supplement to Web-centric standards like ISO/IEC 40500 (WCAG 2.0). The standard was adopted by the according ISO standard and focuses on usability, the importance of context, and the possibilities to individualize applications to one's needs. The target group are individuals with congenital or acquired disabilities, temporary disabilities, the elderly, and situational or environmental barriers.

The ISO 9241-171 standard is part of the broader ISO 9241 multi-part standard, which deals with ergonomics of ICT applications. Many other parts of the ISO 9241 standards series deal with topics relevant to accessibility (e.g., usability, human-computer interaction, presentation of information, design of dialogs and forms).

2.3.4 EN 301 549

In 2014, the European standard EN 301 549 [EN315] was published. This standard aims to provide a guideline for implementing and evaluating accessibility in digital services of any kind (e.g., Websites, software). It contains requirements regarding the accessibility of ICT products and services (hardware, Websites, software, non-Web documents). The minimum requirements for Websites match the level AA of WCAG 2.0.

Besides the EN 301 549, there are also three Technical Reports that help ensuring accessibility in ICT products and services:

- *TR 101 550* - Documents relevant to EN 301 549 (can be used as a reference source for accessibility evaluation)
- *TR 101 551* - Guidelines on the use of accessibility award criteria suitable for public procurement of ICT products and services in Europe
- *TR 101 552* - Guidance for the application of conformity assessment to accessibility requirements for public procurement of ICT products and services in Europe

2.4 Technologies and Their Influence on Accessibility

Accessibility is heavily dependent on the technology being used, since it influences how an application is being developed and how it is used. Some technologies make it easier to create and use accessible technologies, while others can make it harder. Also, the way in which technologies are used can be relevant. Even standards like WAI-ARIA, which were intended to improve accessibility, can render an application inaccessible if not used in the right way.

This section gives a short introduction into the implications on accessibility of technologies which are central to this thesis.

2.4.1 Responsive Web Design

Responsive Web Design (RWD) [Mar10] is a technique for creating Web pages in such a way that they adapt to different browser platforms. Instead of creating separate tailored designs for several kinds of viewing devices (e.g., a desktop computer, tablet, or smartphone), a single design is created and delivered to the browser that provides a customized experience depending on the viewing device, i.e., it will appear differently on a desktop computer than on a tablet or a smartphone. A common analogy used to illustrate this concept is that the Web page content is like water: when poured into a cup, water adapts to the shape of the cup, while on the other hand, when poured into a bottle, water adapts to the shape of the bottle.

To achieve this kind of adaptability, RWD is implemented using fluid grid systems and Cascading Style Sheets (CSS) media queries. A grid system is a layout technique to organize layout elements within a page into a set of columns. This allows for a consistent design in which the elements stay aligned. In a fluid grid system, relative units (like `em`, `rem` or `%`) are used so that the layout can fit into various screen sizes. Media queries were introduced as part of the CSS3 specification [W3Cd]. A media query allows the Web page to use certain style rules depending on specific characteristics of the viewing device, most commonly the browser width, but also the media type (like `screen`, `print`, or `speech`). A media query consists of two parts: the media type and the actual query (e.g., `max-width: 1024px`) and can be used within the `<link />` tag in order to load a CSS file or inside a CSS file where it encloses a set of style rules within curly braces. When the viewing device characteristics match the query, the CSS file (in case of the `<link />` tag), or the enclosed style rules, respectively, are applied to the markup.

Media queries in combination with a fluid grid system allow Web pages to reflow their layout at certain so-called breakpoints, so that, for example, a four column layout is displayed on a desktop computer, while the content reflows into a two column layout on a tablet, or on a single column layout on a smartphone, respectively.

Not only can the grid layout be restructured, but RWD can also adapt font or target sizes (e.g., making them bigger on small screens), selectively show and hide GUI elements (e.g., making the GUI simpler on small devices by hiding less important information), or support device-dependent input modalities (like enabling swiping or pinching on touchscreen interfaces). Importantly, RWD can eliminate the need of scrolling into two directions since the layout can adapt itself in a way that only requires scrolling in one direction.

Studies show that RWD can benefit accessibility in multiple ways [HAS+15] [BS15]. Hallett et al. [HAS+15] found that reading comprehension tasks were completed more quickly and accurately when using a Web page that leverages RWD than when using screen magnifiers (which typically introduce the need of horizontal scrolling). Benda and Smejkalová [BS15] found that participants suffering from mental disabilities (Down syndrome and perinatal encephalopathy) could perform navigation tasks more easily on RWD Web pages than on their non-responsive counterpart.

2.4.2 HTML, CSS, and JavaScript

HTML is a document markup language, which is used to structure Web pages. Its latest version is 5.1, which became a W3C Recommendation on 1 November 2016. HTML 5 introduced many new features, among them the `<audio>` and `<video>` tags, the 2D drawing API of `<canvas>`, many new content structuring tags like `<section>`, `<article>`, `<header>`, `<footer>`, and `<nav>`. All in all, HTML 5 seems to have evolved to be more than just a markup language. The new changes are double-edged, however. On the one hand, they pose a promising chance to enhance accessibility significantly. For example, by introducing the new content structuring tags, screen readers and similar assistive technologies have more semantic information about a certain piece of content and can treat it accordingly. WAI-ARIA can further enhance the accessibility of HTML markup. Another example are the `<audio>` and `<video>` tags, which were introduced to remove the reliance on third-party browser extensions like

Flash[1] or Silverlight[2], which bring their own accessibility issues and are not even available on every device (think of, e.g., Apple devices). On the other hand, some new features of HTML 5 can make accessibility engineering more difficult. The new `<canvas>` element can be used to render interactive bitmap images, like graphs, animations, or even games. These images cannot be interpreted by, e.g., screen readers and, originally, there also were no default focus indicators. The contents of a `<canvas>` element can be described by fallback contents and hit regions have to be defined in order to associate a canvas element with a fallback element. While the `<canvas>` element surely has introduced interesting new possibilities for user interaction, it also introduced considerable effort for accessibility engineering and should be used with caution.

CSS have the major benefit for accessibility to separate document structure from presentation. CSS adds spacing, alignment, and positioning to elements, colors and font sizes can be adapted, and element misuse can be prevented (e.g., by using background images instead of `<img>` tags for decorative elements so that they are not visible to screen readers). Users can override styles with their own style sheets, e.g., for overriding colors when they're colorblind. CSS also provides many attributes to modify a screen reader's voice, volume, pitch, stress and even pauses in speech. Often, CSS is also used to provide visually hidden content like instructional cues and indicators only for screen readers (e.g., a "Skip to main" link at the top of the document or a "You are here" label previous to breadcrumbs).

JavaScript enables dynamic behavior of a Web page, and to load content changes without requiring a page refresh. Typical accessibility problems due to JavaScript are lack of keyboard accessibility, lack of user control, and confusion or disorientation. Using event handlers like `onmouseover` or `onclick` prevent the user from interacting with the application via the keyboard, because the event handlers are not triggered. Lack of user control and confusion can originate from dynamically changing contents, where the user might not be able to undo the actions or does not understand why and what information has changed. But using JavaScript does not render the application inaccessible per se, although it sometimes requires additional effort (e.g., adding WAI-ARIA live regions for dynamic content) and special care (e.g., using input device independent event handlers like `onfocus` or `onchange`). It is also a common misconception that disabled people, especially blind individuals, do not use JavaScript, i.e., have it disabled. A survey conducted by WebAIM in 2014 [WAI14] found that 97,6 % of respondents (all screen reader users) had JavaScript enabled. Nevertheless, it should be noticed that some users might have JavaScript disabled by default or scripting is not fully supported by their user agent (e.g., when using text-based browsers), thus it is recommended that the application is still accessible if JavaScript is disabled or at least an alternative version without JavaScript is provided.

[1]https://www.adobe.com/products/flashplayer.html (Last Access: 2017-01-08)
[2]https://www.microsoft.com/silverlight/ (Last Access: 2017-01-08)

Frameworks and Libraries Introduced to UCP

Several frameworks and libraries have been introduced to UCP as part of this thesis to improve accessibility.

Bootstrap 4.1.2

Bootstrap[3] is an HTML, CSS and JavaScript library which includes several GUI components as well as a grid system that allows for responsiveness. It is a very popular and widely-used library. Although there are other libraries with similar features, Bootstrap was chosen based on the findings of Duarte et al. [DMV+16], who have analyzed the accessibility of Web applications using different technologies and frameworks. Bootstrap seemed to facilitate accessibility better than other libraries.

Bootstrap's most powerful feature is the grid system. It includes a container that holds several rows and up to twelve columns per row. Responsive breakpoints defined by CSS media queries determine at which viewport width elements scale up or down, respectively. Columns are defined by CSS classes and one HTML element can have multiple of these classes. For example, if a `<div>` element has the CSS classes `col-md-6` and `col-sm-12` with Bootstrap's default breakpoint settings, its width will be half the width of its parent element (e.g., the `<body>` element) on a screen width larger than 767 pixels (this could be the screen of a tablet or a desktop computer). If, on the other hand, the `<div>` element will be shown on a screen smaller than that, its width will be the same as the width of its parent. That is, if the parent holds more than that one `<div>` element, these will reflow into another row, since the `<div>` element already takes up all the width. This enables scenarios where a four-column-layout shown on a desktop computer reflows into a two-column-layout on a tablet computer and finally into a one column layout on a smartphone.

In addition, Bootstrap already provides some accessibility features, like the CSS `sr-only` class for hiding elements from the screen while making sure that they are still accessible by screen readers. This is done via absolute positioning and making sure that the contents appear outside the viewport. Some other features, like its default color palette, might lead to accessibility issues (e.g., some colors have insufficient contrast on white background) and need to be taken care of manually. This is where the Assets Framework is useful.

Assets Framework

The Assets 3.4.1 framework[4] is, like Bootstrap, an HTML, CSS and JavaScript library which is based on various other libraries (namely Bootstrap and all libraries listed below except Combobo). It was developed by the Centers for Medicare and Medicaid Services to provide Section 508 compliant GUI components. Even though the Section 508 accessibility standard, which was developed by the American government as part of the Rehabilitation

[3]https://getbootstrap.com/ (Last Access: 2018-11-18)
[4]https://assets.cms.gov/resources/framework/3.4.1/Pages/ (Last Access: 2018-11-24)

Act, is less restrictive than the WCAG guidelines, the Assets framework provides a good baseline for accessibility. It compensates for most of Bootstrap's accessibility issues and gives detailed guidance for integrating and testing of the components into a Web application.

In version 3.4.1, Assets provides 27 components, not all of which are used in UCP. For each component, the documentation provides demonstrations including a basic template for integrating the component into a Web application. Moreover, all dependencies on other libraries are listed. Important notes for developers give instruction of what to pay attention to when using the component (e.g., which sensible values for parameters are necessary to ensure maximum accessibility). There are also notes for testers which explain what features of the component are important to test (e.g., that the screen reader announces day names within the date picker component while navigating through it). Available keyboard operations for the components are also specified. Finally, known accessibility issues for the component are explained in detail, so that developers can focus on overcoming them or are at least noticed about what issues might arise when users use certain kinds of assistive technologies. Some of these issues are the result of some assistive technologies not interpreting certain accessibility markup correctly. For example, JAWS does not announce the full names of days in the date picker but only their abbreviated versions like "Mon", although they full names are marked up with `abbr`, `title`, and `aria-label`. In these cases, later versions and updates of the assistive technologies could solve that problem.

jQuery and Plugins

jQuery is a JavaScript library which is mostly known for its Document Object Model (DOM) traversal and manipulation functionality, but also provides event handling and Ajax as well as basic animation.

jQuery UI[5] 1.12.1 is an extension to jQuery and provides various GUI widgets along with effects and functions to enable certain user interactions (e.g., enabling drag and drop or resizing of elements). It also enables developers to create their own reusable widgets.

The Assets framework's date picker component is based on *Yet Another DatePicker*[6], which provides many accessibility features by making use of ARIA roles and states, presenting the date picker in a semantically correct `<table>` markup, providing extensive keyboard support, and making sure that screen readers read all the important information (i.e., the full date including day names).

The *Custom Input*[7] plugin was leveraged by the Assets framework to provide a more consistent appearance of radio buttons and checkboxes across browsers.

[5]https://jqueryui.com/ (Last Access: 2018-11-25)
[6]https://github.com/freqdec/datePicker (Last Access: 2018-11-25)
[7]https://github.com/filamentgroup/jQuery-Custom-Input (Last Access: 2018-11-25)

The *Input Mask*[8] extension was taken from Jasny Bootstrap 3.1.3, which extends Bootstrap by providing further stylings and JavaScript functions to enhance user experience. Input masks indicate a specific data format to the user by prefilling the input field with format hints (e.g., "__.__.____" for the required date format in a date picker input field). The input mask is replaced by the actual user input one character after the other while the user is typing. This is an advantage over simply using placeholder texts on an input field to indicate the data format, since the placeholder vanishes as soon as the input field is focused.

The *jQuery Validation*[9] 1.17.0 plugin provides extensive functionality for client-side form checking via JavaScript. Developers can define their own form checking rules or use the already provided ones (e.g., for date formatting, number formatting or required fields). The plugin listens to the form submit event and checks the form against all the defined rules. If there are any form errors, the plugin prevents the browser from sending the form data to the Web server and instead inserts the error messages into the DOM. Error messages can be customized as needed to make them as descriptive as possible. As soon as the user has corrected an error, the plugin removes the error message from the DOM, so that the user gets immediate feedback of whether the input matches the expected format. When the form is submitted again and there are no errors, the form data is sent to the Web server.

Accessible Responsive Tabs[10] provides an accessible tab and accordion component. It will append all the necessary ARIA properties to their respective HTML elements (e.g., `aria-controls` and `aria-selected` to the tabs, and `aria-labelledby` and `aria-hidden` to the tab contents). It is also responsive, i.e., it can switch between tab view and accordion view at a certain media breakpoint.

Combobo

Combobo 2.0.0[11] is an accessible combobox component provided by Deque Systems, a company that focuses on digital accessibility, which also developed aXe[12], a popular browser extension for accessibility testing. Combobo is a combined text field and dropdown, i.e., one can either select options by using the dropdown or search for available options via the text input field. The searched text is highlighted within the option. It makes use of all the relevant ARIA attributes (like `aria-owns`, `aria-autocomplete`, `aria-expanded`, and so on). The announcements of screen readers regarding the number of selected items or which item was selected, can be customized.

[8]http://www.jasny.net/bootstrap/javascript/#inputmask (Last Access: 2018-11-25)
[9]https://jqueryvalidation.org/category/plugin/ (Last Access: 2018-11-25)
[10]https://github.com/stevenMouret/accessible-responsive-tab (Last Access: 2018-11-25)
[11]https://github.com/dequelabs/combobo (Last Access: 2018-11-18)
[12]https://www.deque.com/axe/ (Last Access: 2018-11-18)

Modernizr

Modernizr 3.6.0[13] is a feature detection JavaScript library which identifies what HTML, CSS, and JavaScript features are available in the user's browser. Based on this information, progressive enhancement and graceful degradation become possible. Progressive enhancement means adding more technically advanced features to a Web application if the browser supports them. Its premise is to provide a sensible basis which is available on all devices and browsers, and to gradually extend that basis as more features are supported. Graceful degradation, on the other hand, works the other way around by detecting which features are not supported by a device or browser and disabling them or providing alternatives (and thus providing backwards compatibility).

Duarte et al. [DMV+16] found that applications using Modernizr tended to be more accessible than other applications. Modernizr detects features by running quick small tests when the page loads. The results of these tests are stored in a JavaScript object (i.e., `Modernizr`) and appended to the `<html>` tag's CSS classes. This way, the availability of features can be requested via JavaScript by simply testing whether the `Modernizr` object's properties are set to `true`, or reacted to via CSS by styling the corresponding CSS classes (e.g., setting `display: block;` for the paragraph which indicates that not all features are enabled when the HTML tag contains the CSS class `.no-js`). Modernizr also enables to add your own tests at page load (e.g., to test the availability of certain JavaScript frameworks on the Web page).

Require.js

RequireJS 2.3.5[14] provides asynchronous module loading and dependency management for JavaScript code. Its purpose is to load JavaScript files not always at page load but only when necessary (i.e., lazy loading), and hence to improve page loading speed and code quality. It enables the developer to define modules and declare dependencies on them by making use of the Asynchronous Module Definition API specification. It also allows building bundles of combined and minimized JavaScript files to minimize page loading time.

Font Awesome

Font Awesome[15] is an icon font. Fonts made of icons have various advantages over image files, for example, one can change an icon's color easily via CSS or scale its size by changing the font size. This is especially helpful for responsive layouts, where an icon is displayed bigger on a smartphone than on a desktop computer. Icons are shown via CSS classes on an empty tag (e.g., the `<i>` or `<span>` tag). In case the icon is not just decorative but conveys meaning, a `title` attribute can provide a tooltip text and

[13]https://modernizr.com/ (Last Access: 2018-11-24)
[14]https://requirejs.org/ (Last Access: 2018-11-24)
[15]https://fontawesome.com/ (Last Access: 2018-11-18)

the element can be accompanied by a text element that is visible only to screen readers (similar to the `alt` attribute for images).

2.4.3 Mobile Devices

The importance of mobile device accessibility has become evident as the smartphone and tablet industry emerged some years ago. Nowadays, in some parts of the world, especially in developing countries, people are more likely to have access to a mobile device than a desktop or laptop computer [RM08]. In 2015, Google even claimed that more Google searches were performed via mobile devices than desktop computers[16]. Due to the fact that mobile devices have different, and often restricted, prerequisites when it comes to GUI engineering (e.g., small screens, low bandwidth), special care needs to be taken when addressing the accessibility of mobile applications. The W3C has developed various best practices for mobile Web applications [RM08] [CS10]. A working draft from February 2015 also explains how the WCAG 2.0 guidelines apply to mobile development [PSW15].

Mobile users usually have different intentions and interests than desktop users. They often look for a specific piece of information relevant to their current context, for example, finding out the opening hours of a shop or when the next train arrives in their proximity. Mobile users typically are also less interested in browsing and reading lengthy documents. Some UI elements are not suitable for mobile devices. For example, pop-ups can cause confusion because they often appear without explicitly being triggered by user interaction, take away a majority of the screen and sometimes it is not clear to the user how to close them, hence they should be avoided. For applications involving network traffic, especially Web applications which require a lot of navigation between pages, long retrieval times should be avoided.

Because the screen size of mobile devices is typically much smaller than the screen size of desktop or laptop computers, important information should be placed at the top where scrolling is not necessary. It can also be reasonable to hide some detailed information, which is only displayed to users with a larger screen size. Developers also have to consider that there is a wide variety of screen sizes for mobile devices for which the content needs to be tailored. Additionally, applications should support both landscape and portrait orientation, because some users need to use their device in a fixed orientation (e.g., if it is mounted on a wheelchair).

An important difference between mobile and desktop devices is the way users interact with an application. While a desktop user typically uses a physical keyboard and a mouse, mobile users mostly rely on touch gestures and a virtual keyboard. Assistive technologies are available for both desktop and mobile devices. The virtual keyboard of a mobile device is usually very small and takes away a certain, often considerable, amount of screen space. It also provides barely any haptic feedback, i.e. one often has only visual feedback of which key was activated (if any) while on a physical keyboard one can feel the boundaries of each key and really needs to push it in order to activate it. In

[16]https://adwords.googleblog.com/2015/05/building-for-next-moment.html (Last Access: 2017-01-13)

addition, mobile users typically use only one or two fingers to type. Due to these reasons, keystrokes should be minimized by avoiding free text entry and using default values for input fields where possible. Hyperlinks should be as short as possible and automatic sign-in should be provided to avoid the cumbersome input of passwords. Targets for touch gestures have to be large enough and should be surrounded by a certain amount of inactive space, so that the user does not accidentally activate the wrong target. Features like mouse-over effects do not work for touch gestures. Gestures should be as easily as possible to perform. Especially users with motor impairments or using a screen reader or other assistive devices (e.g., head-mounted display, stylus) can have difficulties with multi-touch gestures (i.e., gestures involving multiple fingers simultaneously). The same holds for other control options triggered by physically manipulating the device (e.g., tilting, shaking). Hence, although these options can be supported by the application, keyboard or simple touch alternatives should still be provided.

2.4.4 Assistive Technologies

Assistive technologies alter the way a user interacts with an application. For example, some input devices for users with motor impairments (e.g., head-mounted pointers or eye-tracking) make it hard for them to perform complex touch gestures or key combinations. It is important to provide alternative ways of interacting with the application that do not require such complex control options, e.g., by displaying buttons that can be clicked or activated via a single key. Screen magnifiers have difficulties enlarging texts that are embedded in graphics, since it becomes pixelated and thus hard to read.

One assistive technology that is probably the most complex to address is the screen reader. This is because most operating systems and applications are designed for visual feedback (i.e., GUIs that the user interacts with). A screen reader's purpose, on the contrary, is to transform all the visually presented information into acoustic and/or haptic information, i.e., it addresses a completely different sense. The most important aspect for screen readers is for the application to provide properly structured contents and suitable semantic information along with the GUI elements. A screen reader does not just read everything from top to bottom, it is also capable to navigate through special contents of an application. For example, it may just read the headlines of a document, or only the navigation elements, such that the user can quickly get an overview of the contents and functionality an application provides. The majority of screen reader users does not use a mouse, which is why an application needs to support keyboard-only interaction. Typically, the screen reader has configured a primary language, which often matches the language of the operating system. It is important that the application indicates the language of its contents, so that the screen reader can switch to the right language. Another aspect to consider are images, since a screen reader cannot describe them if no alternative text describing their meaning is provided.

The way in which assistive technologies access the GUI of an application has evolved over time. Originally, heuristic techniques were used to determine the purpose and functionality of user interface elements on the screen by using hooks into graphics calls and scraping the screen. For example, labels organized in a horizontal row at the top of the application could be interpreted as a menu. Based on the results, an alternative off-screen model, which can be seen as a snapshot of the screen at any given time, was built. These heuristics were very complex and needed to be updated whenever the GUI changed dynamically. These updates were time-consuming and the results were not always accurate if the information presented on-screen was ambiguous.

In the late 1990s, instead of letting the assistive technologies interpret the GUI elements, operating systems and software started to expose information about GUI objects and events through an Application Programming Interface (API). Assistive technologies can query these so-called accessibility APIs to get relevant semantic information (such as names, roles, and values) about the user interface elements and interact with them. Accessibility APIs are a more reliable way for assistive technologies to access the GUI than off-screen models, since developers can control which information is exposed by each element within the application. *Microsoft Active Accessibility (MSAA)*[17] was the first available accessibility API. Today, *UI Automation (UIA)*[18] for Windows, *IAccessible2*[19] for Windows and Linux, *Assistive Technology Service Provider Interface (AT-SPI)*[20] for Linux, and the *NSAccessibility*[21] protocol for Mac OS X are popular examples.

Accessibility APIs expose information about an application's GUI via the accessibility tree, a hierarchical representation of accessible objects that the GUI consists of. An accessible object is a GUI element (e.g., a dialog, menu, container, button, label, checkbox, input field), which exposes semantic information like its name, role, state, or value that can be used by an assistive technology. Based on this information, the assistive technology then builds an alternative user interface which is presented to the user (like a sequence of Braille signs or speech in case of a screen reader). In case the assistive technology supports alternative ways of interacting with an application (e.g., using voice commands instead of mouse clicks or key strokes), it communicates these user interactions through the accessibility API back to the application, where they are translated into appropriate actions in the context of the original user interface.

In case of Web applications, the accessibility is maintained as a parallel structure to the DOM, a tree structure that represents the contents of a Web page. The DOM is used to render the Web page's contents and to manipulate them via scripts. Basically, the accessibility tree is a subtree of the DOM, which trims out elements not relevant to assistive technologies in terms of performance and simplicity. Such elements include e.g., `<div>` or `<span>` elements, which are typically used for introducing style changes. Sometimes, these elements are used to represent interactive user interface elements (e.g., dialogs or regions which contain dynamically changing contents). In these cases, WAI-ARIA can be used to add semantic information (like a role or live region, see

[17]https://msdn.microsoft.com/en-us/library/ms971350.aspx (Last Access: 2017-01-25)

[18]https://msdn.microsoft.com/en-us/library/ee684013%28VS.85%29.aspx (Last Access: 2017-01-25)

[19]https://wiki.linuxfoundation.org/accessibility/iaccessible2/start (Last Access: 2017-01-25)

[20]https://developer.gnome.org/libatspi/stable/ (Last Access: 2017-01-25)

[21]https://developer.apple.com/reference/appkit/nsaccessibility-jku (Last Access: 2017-01-25)

Section 2.3.2), which then results in the inclusion of the `<div>` or `<span>` element in the accessibility tree. Elements can thus be modified to enhance accessibility.

2.5 Accessibility Evaluation

In order to check if an application is accessible and fulfills the requirements of standards like WCAG or legal regulations, a structured accessibility evaluation process is needed. An inspection of the application can be performed manually, semi-automatically, or automatically. Usually, a combination of these techniques is used for accessibility evaluation, since automatic tools are not able to detect every possible accessibility issue. For example, automatic tools cannot decide if the structure of the content or the tab order make sense or if an alternative description matches the image it belongs to. Also, automatic tools fail to check dynamically changing contents for their accessibility. Sometimes, automatic tools detect *possible* violations that need to be checked by an evaluator. Hence, automatic accessibility evaluation tools can bee seen as a way to assist accessibility evaluation for identifying accessibility problems which can be easily detected by software, but a manual inspection of the application is still necessary.

When performing a manual accessibility evaluation, the evaluator confirms or refutes accessibility problems found by automatic tools and also discovers new accessibility issues which could not be found via automation. Typically, the evaluator uses existing guidelines and standards like WCAG, during this process. In a semi-automatic approach, the evaluator also uses tools like browser add-ons to investigate accessibility issues. A common way to test accessibility using assistive technologies is to perform a screen reader accessibility evaluation. Here, the evaluator uses screen reader software like JAWS[22], listens to the output and tries to comprehend the content and navigate through the application using only the keyboard.

In addition to checking an application against standards and legal regulations, performing tests with real users can be beneficial. Such tests can lead to a better understanding of accessibility issues and how disabled individuals use an application. However, performing tests with users can easily become time-consuming, and covering all the different types of disabilities is hard to achieve. The extent to which such tests are performed depends on the size and purpose of the application. For example, a governmental application needs to be tested more thoroughly than a small private software tool that is only used by a small number of users whose abilities and circumstances are known to the developers.

Bai et al. [BMSF16] identified four different accessibility evaluation methods, which, apart from (1) automatic and semi-automatic testing and (2) user testing, also include (3) expert testing and (4) testing using simulation kits. They recommend to use at least two of these methods. In expert testing, an expert performs the evaluation using a set of heuristics. Sometimes, persona walkthroughs, in which an expert simulates a persona while performing predefined tasks, or cognitive walkthroughs, where the expert elaborates

[22]http://www.freedomscientific.com/Products/Blindness/JAWS (Last Access: 2017-01-26)

his or her mental process while performing certain tasks with the application. Expert testing is similarly time-consuming as user testing. Simulation kits are wearables that simulate aspects of an impairment like glasses mimicking vision impairments or tools to simulate hand tremors. They do not require expert knowledge to gain valuable insight into possible accessibility issues, but they involve noticeable upfront costs.

2.5.1 Evaluation Methodologies

Website Accessibility Conformance Evaluation Methodology (WCAG-EM)

Various methodologies have been introduced for evaluating an application's accessibility. The WAI published a Working Group Note regarding a WCAG-EM in July 2014. This methodology introduces an evaluation procedure for Websites, comprising five steps, which is shown in Figure 2.1. Evaluators proceed each step sequentially. If new information comes up during the evaluation process, they may return to any preceding step.

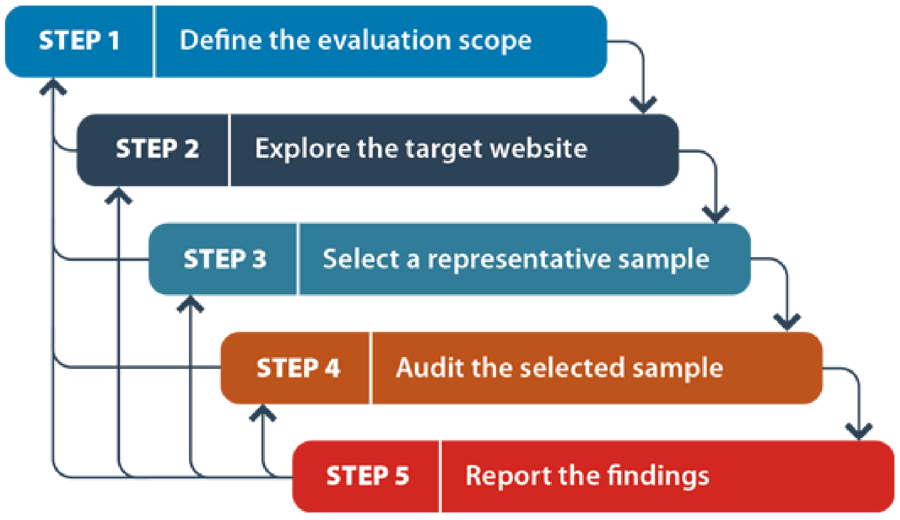

STEP 1
Define the evaluation scope
STEP 2
Explore the target website
STEP 3
Select a representative sample
STEP 4
Audit the selected sample
STEP 5
Report the findings

Figure 2.1: The five steps of the WCAG-EM 1.0. Copied from [VAZ14].

In the first step, the evaluation scope is defined. The parts of the Website considered in the evaluation are defined, including mobile and language versions as well as potential third-party services used by the Website. Then, the target conformance level of the WCAG is defined. Supported browsers, assistive technologies and other user agents are determined as well. Analyses of use cases or target user groups could be included here. If user tests should be performed, they are planned during this first step.

The second step involves the exploration of the target Website to gain an understanding of its purpose and features. Its features and types of Web pages (in terms of styles and layout, content types, functional components, and so on) are identified. All used technologies have to be identified in order to know which guidelines need to be applied.

During the third step, a representative sample of Web pages is selected from the Website. The sample is representative, if it *"reflect[s] the accessibility performance of the [W]ebsite with reasonable confidence"*. If it is feasible to evaluate the entire Website, this step can be skipped. The sample should consist of a structured sample, which includes all Web page types identified in the previous step, and a randomly selected sample which is 10% of the size of the structured sample and is used to check whether the sample is sufficiently representative for the Website's accessibility performance. It is important to include complete processes (which are defined, for example, in use cases) in the sample.

In the fourth step, the selected sample (or the entire Website, if feasible) is audited, i.e., evaluated in detail. For each Web page and each complete process, it is checked if it adheres to the conformance requirements of the target conformance level of the WCAG. Finally, the structured and the randomly selected sample are compared. The randomly selected sample should not contain any new content types or show outcomes which were not found in the structured sample, otherwise the evaluators need to return to step three to include more Web pages that include the new content types or outcomes.

The last step comprises of reporting the evaluation findings. The findings are documented throughout the process in every single step and are aggregated and summarized in this last step to produce the final report document. Evaluation specifics like used evaluation tools and methods are included in the report. Optionally, a machine-readable report, using the Evaluation and Report Language (EARL)[23], can be provided.

Unified Web Evaluation Methodology (UWEM)

The EU Web Accessibility Benchmarking Cluster proposed the UWEM with the goal to provide standardized methodology for evaluating the accessibility of Websites in the EU. Although this methodology would be very helpful in testing the compliance of a Web application to the EU regulations, its latest version was published in 2007 and addresses WCAG 1.0, which is outdated at the time of writing this thesis. There was an intent to migrate the UWEM to incorporate WCAG 2.0, but the migration plan was published back in 2008. By the time of writing this thesis, no updated version of UWEM has been published, hence, this methodology can be considered outdated.

Methodology for Heuristic Evaluation of Web Accessibility Oriented to Types of Disabilities [OTD16]

In 2016, Orozco et al. [OTD16] proposed a methodology that was designed as a complement to the WCAG-EM process, which they claim does not take into account the particularities of each form of disability. Their methodology comprises the following five steps:

[23]https://www.w3.org/WAI/intro/earl (Last Access: 2017-01-26)

1. *Analysis and Characterization of the Population:* In this first stage, the various kinds of disabilities and their needs and barriers are identified with the help of literature reviews describing such characteristics.
2. *Definition of Indicators for Evaluation:* The characteristics found in the first stage are now used to identify specific barriers the different groups can face when accessing an application. Using WCAG and similar guidelines, the indicators for accessibility issues are picked and categorized for each form of disability identified.
3. *Definition of Heuristics:* During this stage, heuristics are defined considering the indicators characterized before, which include examples for meeting or not meeting the criteria.
4. *Implementation of Accessibility Evaluation:* This stage involves identifying the scope of the evaluation (e.g., which parts of the application). Orozco et al. advise to have a group of evaluators with varying experience with and knowledge of the specific barriers being addressed.
5. *Analysis of Results:* At this final stage, the results from the evaluation are analyzed and measures for solving the occurred accessibility issues are developed.

The purpose of this methodology is to address the peculiar problems of people with specific disabilities and evaluating an application from this particular viewpoint, instead of just checking for general principles.

2.5.2 Tools

Using accessibility evaluation tools can help in determining the conformance to standards and legal regulations for accessibility checks which can be executed automatically (e.g., checking if every input field is labeled) and assist the evaluator in assessing accessibility aspects which need to be evaluated manually. However, these tools cannot assert that an application is accessible, or even compliant. Human judgment and manual evaluation is always necessary as well.

Accessibility evaluation tools can and should be incorporated into every step of the development process. Designers can use them, for example, to simulate various types of vision impairments to see if font sizes or color contrasts are sufficient. Developers and content authors can utilize tools which are integrated into their development environment (e.g., editors, content management systems) to discover and avoid accessibility issues early on. Testers and evaluators use these tools to examine the overall accessibility of the application in detail, and give feedback to the designers and evaluators.

When choosing suitable accessibility tools, there are various factors to consider:

- *Development Process:* Different roles (e.g., designer, developer, tester) require different tools. Some applications need a higher conformance level to standards and regulations, because they are, for example, governmental applications.

- *Size and Complexity of the Application:* The approach to evaluate larger applications is different from evaluating smaller applications (e.g., automated testing of the whole application and manually testing only certain aspects instead of manually evaluating the complete application).
- *Knowledge, Experience and Skills of the Evaluators:* Accessibility evaluation tools require a varying amount of knowledge about accessibility or the technologies to be evaluated. For example, so-called HTML and CSS validators require knowledge about HTML and CSS, respectively.
- *Development Environment:* Sometimes, it is preferred to integrate accessibility evaluation tools into editors or content management systems.
- *Platform:* While there are online services or browser extensions for evaluating Web applications, there are also tools which need to be downloaded and installed. Some tools are only available on certain operating systems.
- *Scope:* Some tools only check one page or feature at a time or they are limited to certain accessibility issues (e.g., color contrasts). Other tools can evaluate large parts or the complete application at once.
- *Usage:* Accessibility evaluation tools differ in the way they are used. There are, e.g.:
  - *Fully Automatic Tools*: Some tools do not require any user interaction. The evaluator just provides the application files, starts the tool and gets back the results. In some cases, the issues found by such tools need to be validated.
  - *Wizards:* Some tools guide the user through the evaluation process step-by-step, which can be useful for manual validation of the issues found automatically. These tools also aid the evaluator in checking certain issues manually, e.g., images are shown to the user and the user assesses the appropriateness of their textual descriptions.
  - *Transformations:* There are also tools which transform the application, for example, by showing only textual contents, removing the colors or read the contents aloud.
- *Reporting and Presentation of Results:* There are various types of applications that differ in the way they present the results and generate reports. For example, they provide:
  - *Textual Reports:* These tools generate lists describing the individual accessibility issues and sometimes also include examples of possible solutions. Often, they can be customized to one's needs, and are fully automated.
  - *Integrated Feedback:* Some tools highlight accessibility issues directly in the GUI. This is helpful to immediately recognize the parts of the application which need to be improved.

- *Machine-Readable Reports:* Sometimes, accessibility evaluation tools provide a way to output their results in a machine-readable format like EARL. Results from various tools could be integrated in this way into a final, aggregated report.

Usually, it is best to use a variety of different tools to cover as many accessibility issues as possible.

2.6 Accessibility and the Software Development Process

Since there is such a great diversity of disabilities in using ICT, making it accessible can be a complex task, which also depends on the type of content, the size, complexity, and purpose of the application as well as the used technologies. To achieve good accessibility, it is important to integrate accessibility engineering into every stage of the development process [BMSF16] [GK09] [XFW07]. Trying to fix an inaccessible application after development has finished can pose a considerable effort, especially if developers missed to adhere to current standards or the application involves lots of multimedia content and complex user interface controls.

Accessibility cannot be fully automated [XFW07]. In the same way as evaluating accessibility cannot be performed purely automatically because it requires human judgement for certain issues, accessibility engineering needs manual effort for some aspects. For example, designers and model engineers need to provide good usability (e.g., by making the flow of interactions intuitive), and multimedia contents have to be transcribed or supported by subtitles. However, by integrating accessibility considerations into the automated generation of GUIs, at least some aspects of accessibility can be addressed right from the beginning of user interface development. It can also reduce development time and prevent implementation errors by providing correct implementations of accessible user interface controls.

2.7 Automated GUI Generation

The concept of *model-based* software development promotes the usage of models, i.e. an abstract representation of the application to be built (or some aspect of it), to aid the software development process. In this approach, models are used for communication between involved parties (e.g., customers and developers), but they were not applied to automatically generate actual code from them [BCW12].

An advancement of this concept is *model-driven* software development, where the models become primary artifacts of the software development process. Using automated transformations, source code is generated from the models [Sch06] [BCW12]. The advantages of this approach are manifold. Following the principle “model once, build everywhere” [BCW12], model-driven development improves reusability and portability.

Once the model is created, it can be translated into source code for a variety of technologies and platforms using different transformation rules automatically. The approach also turns the focus on business problems by empowering domain experts, since modeling can be done without requiring knowledge in programming. Models also serve as an up-to-date documentation of the software being built.

2.7.1 Model Driven Architecture (MDA)

To enforce model-driven development in the software development process, the Object Management Group (OMG) published the MDA in 2003, which provides a set of standards and guidelines [MDA14]. The key approach is to specify higher-level models, which are used to iteratively generate lower-level models and eventually source code.

The highest level of abstraction is provided by the *Business or Domain Model*, historically called Computation Independent Model (CIM). This model represents "real things" and is computation- and platform-independent. It specifies the application's requirements, describes what it is expected to do and which objects and entities form its domain.

One level below is the *Platform Independent Model (PIM)*, which defines details of the structure of a system, while not taking into account the specifics of a certain platform. The PIM is then transformed into a *Platform Specific Model (PSM)*, which provides all the necessary information for generating platform-specific source code.

Finally, the *Implementation* (in the form of source code) is derived from the PSM.

2.7.2 The CAMELEON Reference Framework

MDA is a generic approach for model-driven software development in general, i.e., it can be used for creating various platforms like programming languages, operating systems, databases, middleware-solutions, or user interfaces [Tru06]. The CAMELEON Reference Framework (CRF) [CCT02] provides a standardization of the model-driven development approach tailored to user interface generation. It contains four levels of abstraction, which roughly correspond to the levels described in the MDA [CCT$^+$03]:

1. *Tasks & Concepts Level:* This high-level model specifies interactions between the user and the application and their temporal relationship as well as the domain of interest.
2. *Abstract User Interface (AUI) Level:* Models on the AUI level are modality- and device-independent, but define abstract interaction components and outline how the application will roughly be structured in the end.
3. *Concrete User Interface (CUI) Level:* The CUI model specifies the particular interaction components (e.g., widgets), which already define the appearance of the application, but are still independent from any user interface toolkit. Each target platform (e.g., smartphone, tablet, desktop computer) has its own CUI model.

Hence, models on this level are already modality- and device-dependent, but not yet transformed into actual code.

4. *Final User Interface (FUI) Level:* This level finally contains the resulting GUI source code, which can be compiled or interpreted.

2.8 The Unified Communication Platform

The UCP [PKR12], developed by the Institute of Computer Technology at TU Wien, can be used to create applications for human-machine or machine-machine interaction using a model-driven approach. These applications are tailored to various target devices (e.g., smartphones, tablets, desktop computers) and their models can be created without requiring profound knowledge in programming, which enables designers to create the application's GUI and domain experts to define its business domain. UCP is available as a Rich Client Platform (RCP) application based on the Eclipse[24] Integrated Development Environment (IDE).

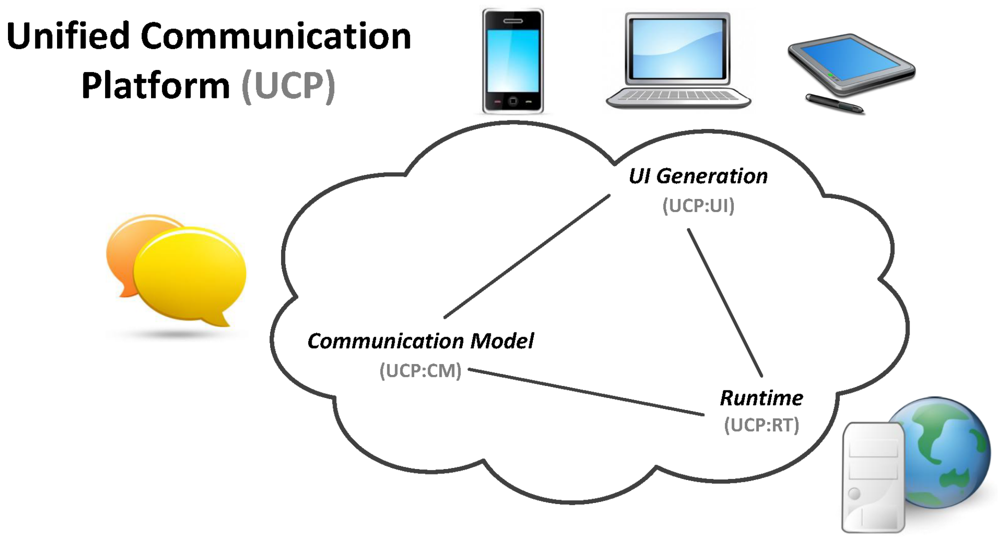


Figure 2.2: UCP comprises of three major components. Copied from [PKR12].

Figure 2.2 shows the three major components UCP comprises of. The *Communication Model (UCP:CM)* defines the communicative interaction between two parties based on Speech Act Theory [Sea69]. The *UI Generation (UCP:UI)* framework semi-automatically generates a Window, Icon, Menu, Pointer (WIMP) GUI from the Communication Model. The *Runtime (UCP:RT)* component uses a Model-View-Controller (MVC) pattern to integrate the generated GUI with the application back-end containing the business logic.

[24]https://eclipse.org/ (Last Access: 2017-01-31)

2.8.1 The GUI Generation Process

As the CRF suggested, UCP uses multiple abstraction levels for generating the GUI. The highest level of abstraction is given by the Communication Model, which is defined by a model designer (i.e., a person who creates the model using UCP). This model is then transformed into lower-level models until the device- and GUI toolkit-specific source code can be generated. Figure 2.3 depicts the various models involved in the UCP GUI generation process and their correspondence to the levels proposed by CRF.

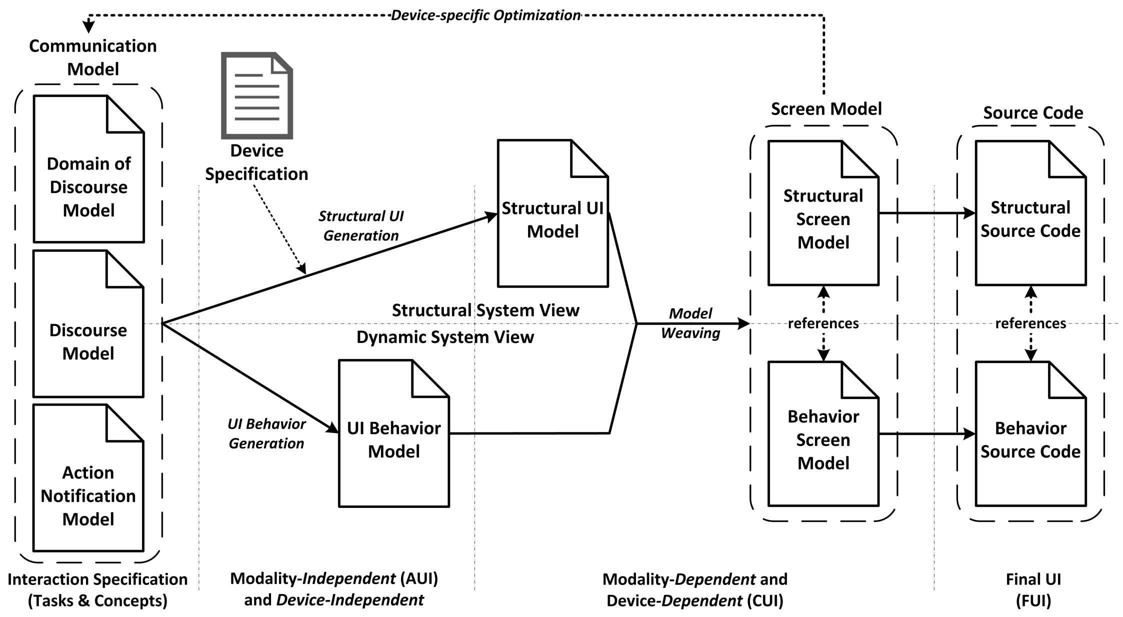


Figure 2.3: The various models involved in UCP and their relationship to the CRF. Copied from [PKR12].

Communication Model

The Communication Model forms the basis for the GUI generation and corresponds to the Tasks & Concepts level of the CRF. It consists of three models, which are created by the model designer and describes the communicative interaction between two parties (human-machine or machine-machine) as well as the business domain involved in it [Pop12].

The *Domain-of-Discourse (DoD) Model* specifies the domain of the interaction, i.e., the entities which the two parties are “talking about” during their communication. It is expressed by an Unified Modeling Language (UML)[25] or Ecore[26] class diagram.

[25]http://www.uml.org/ (Last Access: 2017-02-03)

[26]https://wiki.eclipse.org/Ecore (Last Access: 2017-02-03)

The *Action-Notification Model (ANM) Model* expresses which actions on the entities of the DoD can be taken during the interaction. It provides an interface to the application's business logic. A basic set of actions and notifications are provided by UCP, which can be extended for providing specific functionality for the application.

The *Discourse Model* specifies all possible flows of communication in the application [KPR12]. It is defined by using a Domain Specific Language (DSL) that is based on language theory concepts, which makes it much more intuitive to use for its special purpose than general programming or modeling languages. The model's basic building blocks, the *Communicative Acts*, are based on Searle's Speech Act Theory [Sea69]. Communicative Acts specify elements of speech to express a certain intention: A statement is made to provide information, a question is asked to get information, and a command is given to trigger some kind of action by the other party. Communicative Acts refer to the entities of the DoD and represent basic units of communication. Another important concept of the Discourse Model are *Adjacency Pairs* based on Conversation Analysis [LFG90], which form pairs of Communicative Acts to model typical turn-takings in a conversation (e.g., question and answer, offer and acceptance).

The elements of a communication, i.e., Communicative Acts and Adjacency Pairs, are linked using *Discourse Relations*, which can be of one of two types: *Procedural Constructs* or *Rhetorical Structure Theory (RST) Relations*. Procedural Constructs are used to define the control flow of the communication similarly as programming languages do using loops, if-else-constructs or sequences of commands. Examples of Procedural Constructs are *Sequence*, *Condition*, and *IfUntil*. RST Relations are based on Rhetorical Structure Theory [MT88], and focus on the functionality of text by describing relationships, effects, and constraints of text portions (i.e., Communicative Acts or Adjacency Pairs). There are five RST relations applied in UCP: *Background*, *Circumstance*, *Elaboration*, *Joint*, and *Result*. Together, Communicative Acts, Adjacency Pairs, and Discourse Relations form a tree structure that represents the communicative actions performed by the two parties.

Structural UI Model and UI Behavior Model

Using *Model2Model Transformation* and a set of *Transformation Rules*, the Communication Model is transformed into a *Structural UI Model*.

The Structural UI Model is an abstract representation of the GUI and already contains information about the GUI's structure and content. It is thus already modality- and device-dependent, but still independent from the GUI toolkit being used for the final GUI implementation. Hence, the Structural UI Model corresponds to the CUI level of CRF. During the transformation from the Communication Model into the Structural UI Model, a device specification is used for each target device, resulting in one Structural UI Model per device. The device specification specifies, e.g., the resolution and dots per inch (dpi) of the device, as well as the GUI toolkit and, if the device is operated using a mouse or touch gestures. Using this information, the Structural UI Model is already tailored to the screen size of its target device and places GUI widgets in a space-saving way optimized for it. Widgets are organized in a hierarchical tree.

Besides the structure of a GUI, its behavior also has to be defined. Hence, a *UI Behavior Model*, which represents a state machine, is automatically derived from the Communication Model. This model is independent from the modality and device being used, corresponding to the AUI level of CRF.

Screen Model

Via model *weaving*, the Structural UI Model and the UI Behavior Model are transformed into the Screen Model on the CUI level of CRF. Weaving is the process of associating elements which represent the same aspect of a UI from different viewpoints, from both models with each other to form a single model. The resulting Screen Model consists of two parts referencing each other: the *Behavior Screen Model* (a UML state machine specifying all the possible sequences of screens) and the *Structural Screen Model* (defining each concrete screen with all its GUI elements).

In the end, the final GUI implementation, i.e., the source code, is generated from the Screen Model.

2.9 Related Work

Research regarding the integration of accessibility into model-driven software development approaches is sparse. Accessibility research seems to focus mostly on integrating accessibility measures into the process of programming and especially on the evaluation of accessibility. Some studies focus on integrating accessibility into the modeling process, but address only a specific part of accessibility or discuss the topic without ever providing a prototypical implementation. Some of the studies use the outdated WCAG 1.0 guidelines. Further, many models have not been used in realistic scenarios or day-to-day use due to their complexity and difficulty to implement [JV17]. The scarcity of research in this field indicates that there is still a major potential for scientific work regarding automated generation of accessible GUIs.

In the following, four research trends regarding accessibility of model-driven applications are exemplified.

2.9.1 Modifying Already Existing Applications Using a Model-Driven Approach

There have been several studies that focus on making existing applications more accessible by using a model-driven approach.

Yesilada et al. [YHGS03] developed DANTE, a semi-automated tool, which transforms Web pages that lack sufficient HTML markup (e.g., that only use color and font size to indicate headlines or use image links without alternative texts for navigation) to make them easier to navigate for users with visual impairments. They applied physical traveling metaphors to Web navigation patterns and created an ontology, which is used to annotate Web elements with these metaphors. To identify Web elements which need to be annotated, the so-called travel objects, they developed a framework proposing a semi-automated tool, which can automatically extract and annotate travel objects in a first step. In a second step, these travel objects can be refined by human intervention to gain a better result. When the travel objects have been extracted, they can be used to transcode the Web page automatically to be accessible for users with visual impairments. Various transcoding techniques are proposed by Yesilada et al., e.g., providing a table of contents based on headings and sections, eliminating repetitions by providing "skip links" to the main content or removing repetitive elements, or indicating the size of a certain object (e.g., by adding the text "This item contains two elements.").

Linaje et al. [LLTPT$^+$11] combine the RUX method [LPSF07], a model-driven approach for generating Rich Internet Application (RIA)s, and the SAW (System for Accessibility to the Web) project [SF$^+$07], which provides ontologies for Web accessibility. The approach *"allows enriching already developed Model-Based Web applications with RIA features and accessibility issues".* To do this, they developed the ontology OntoRUX, which is based on WAI-ARIA and defines which attributes RIA widgets have to implement to be accessible. It also checks for possible inconsistencies to keep the widgets' accessibility properties consistent. An example would be a `textbox` component having both the `required` and the `hidden` attribute set to `true`. Consistency can also be checked for changes over time (e.g., if a widget's state and appearance changes when clicking on it) by analyzing representations of the widgets as Linear Time Logic formulae.

While these approaches might enhance the accessibility of already existing applications, they do not support developers to create accessible applications upfront. In addition, DANTE [YHGS03] is only focused on visually impaired users.

2.9.2 Design-Time Approaches

Various studies have focused on how to integrate accessibility concerns into the model-driven engineering process at design-time.

Göhner et al. [GKJ$^+$08] present a model-driven approach of integrating accessibility into GUIs for IT and automation systems, which is derived from the CRF framework. They argue that accessibility is a quality of the user interface which also influences an application's structure and functionality and that less extra effort is required when accessibility is addressed right from the start of the design process. Hence, a so-called user-oriented approach is needed, which starts the design process with analyzing the requirements of the user interface from a user's perspective. Pre-modeling activities like Systematic Layout Planning can be used to discover mental models of potential users. The Tasks & Concepts Level in their approach consists of a Task Model, a User Model, and a Use Model. The Task Model describes all tasks a user can carry out, independently from any possible (personal or environmental) barriers. The Use Model extends the Task

Model by describing whole workflows independent from used technologies or platforms. The User Model describes users and their potential needs and restrictions as well as which tasks they can perform. From the AUI Model, a Navigation Model is derived to gain better insight into the user interface structure. Göhner et al. also include a Context Model describing the influence of various platforms on the application. The CUI Model then is extended to contain semantic annotations regarding accessibility, which are then mapped on WAI-ARIA roles within the Final User Interface.

Vieritz et al. [VJP11] proposed a model-driven, user-centered approach of creating accessible user interfaces, where information about sensoric, motoric and technical disabilities are integrated into the models. Since they focus on the accessibility of e-learning platforms, where the main task is not only to present information to the user but also to impart knowledge, they claim that accessibility has to consider the user's way of thinking and how he or she understands ideas and concepts, too [VPJ07]. For example, a person who is blind by birth has different concepts about the world than a person who has functioning vision or who went blind later in life. Hence, they propose integrating descriptions of a user's mental models into the modeling process in the form of so-called User Models. They also point out that semantic meanings and relationships could be expressed through the model and, hence, this knowledge is made explicit and formal [VJP11]. Their argument is based on the BeLearning project [DJSV05] [VPJ07] [JPV09], which combines semantic encoding (i.e., explicit formulation of relevant knowledge) and model-driven development to enhance the accessibility of e-learning platforms. For example, they model the semantic information for an image, which indicates its functionality for the Web application (e.g., if it is used solely for decorative purposes or if it actually carries important information). Based on this information, an automatically generated HTML `<image />` tag for this image might contain a descriptive `alt` property or not. For enhancing the navigation in the BeLearning project, they proposed using an OOWS Navigational Model [FPAP03] extension for adaptive navigation based on User Models introduced by Rojas et al. [RPF05].

EGOKI [AAC+11] automatically generates accessible GUIs for ubiquitous services, i.e., services that provide unified access via mobile devices to, for example, ATMs, vending machines or ticketing machines. Based on both service characteristics specified by a User Interface Description Language (UIDL) and user capabilities modeled as an ontology, EGOKI's Resource Selector Module chooses the most appropriate representation of an interaction element. This requires the service provider to offer sufficient alternatives for each of the capabilities, since EGOKI cannot create them by itself. To support service providers with creating the service models that are used as input for EGOKI, Miñón et al. [MMA13] developed SPA4USXML, a GUI tool for creating Task, AUI and multimedia resource models. The multimedia resource models describe the various alternatives for interaction elements. A wizard is provided to select the service description files, which are then transformed into tasks for the Task Model, which can then be further refined by the model designer. The Task Model is then transformed into the AUI model. SPA4USXML's AUI editor then enables the user to link abstract interaction elements with the various alternative representations, which are provided by the user. Related to this research is a study by Miñón et al. [MMMA14] regarding the integration of accessibility requirements into UsiXML.

There has also been the idea of integrating accessibility into the model-driven development process as an aspect-oriented concern [MRCG10] [MSMG12]. In this way, accessibility is closely related to architectural concerns, but still treated independently, emphasizing the *"non-functional, generic and cross-cutting characteristics"* [MSMG12] of accessibility. The connection between architectural decisions and accessibility concerns is defined by so-called integration points of varying granularity into the User Interaction Diagram (UID) [VSDS00]. Accessibility requirements according to the WCAG 1.0 guidelines are modeled using a Softgoal Interdependency Graph (SIG) template. They are understood as softgoal concerns as defined by Chung and Supakkul [CS04], i.e., since accessibility is a non-functional requirement there have to be more loosely defined criteria.

The common aspect of the above research is that they incorporate users' capabilities into the models to derive a UI that meets their needs. This requires additional modeling effort and a model designer's in-depth knowledge of a multitude of disabilities.

2.9.3 Guidance and Assistance of Developers

Another approach has been to assist the developers and model designers in creating accessible applications by guiding them and requiring them to specify information that is needed to create accessible GUIs. In addition, technical aspects of accessibility guidelines can be directly incorporated into the GUI components.

De Oliviera et al. [dOFP+14] developed the Homero Framework based on PHP to support the development of accessible Web interfaces. The framework consists of several PHP classes representing user interface components (e.g., `Image`, `Link`, `Table`) which produce accessible HTML code. When invoking methods on these classes, the PHP developer has to provide all necessary accessibility data (e.g., alternative texts for images), otherwise the framework presents a warning to the developer. The automatically generated HTML code adheres to the AAA conformance level of the WCAG 2.0 success criteria. However, only technical aspects of the WCAG 2.0 guidelines which can be detected automatically, are addressed within the framework. Success criteria which involve human evaluation are not being addressed. Since it is a PHP framework, it also can be used only by programmers, not by application designers or domain experts. The framework is intended to avoid common, automatically detectable accessibility mistakes made by developers.

Moreno et al. [MMR08a] created a domain-specific metamodel, the Accessibility for Web Applications (AWA)-Metamodel, which models technical aspects of WCAG 1.0 requirements at the CIM level of MDA. The AWA-Metamodel contains abstract constructs of Web elements including their required attributes (e.g., an alternative text for an image, a level of a heading, or the language of a Web document). To integrate the metamodel into the MDA models, a graphical AWA-Metamodel editor is provided. Finally, a code generator is proposed, which should translate the accessibility concepts from the AWA-Metamodel into accessible HTML code using Model-to-Text transformation. The AWA-Metamodel does not aim to model aspects requiring human assessment, but still covers around 50% of the WCAG 1.0 checkpoints. While not obtaining full accessibility, integrating accessibility requirements into a formal model which could be used to automatically generate the GUI still would improve accessibility.

As part of the Accessibility for Web Applications research proposal, Moreno et al. [MMR08b] also suggested an editing tool for creating accessible Web contents. The editing tool should use accessible templates in XML format, which can be edited and filled with content by the user and are then transformed via XSLT into accessible XHTML code. This should guide and support the user when creating accessible content, for example, requiring him or her to provide mandatory information (e.g., an alternative text for images) or providing a dictionary module for automatically extending abbreviations.

González-García et al. [GGMM15] developed a model-based graphical editor for designing accessible media players. It leverages UsiXML[27], a UIDL. The authors modelled relevant accessibility requirements as tasks and relationships in the Task Model of UsiXML [GMM15] and developed a graphical editor for users with no prior experience in accessibility [GGMM+13].

2.9.4 Run-Time Approaches

Several studies have focused on run-time approaches and context-aware adaptive GUIs which change their appearance in response to certain user behaviors and preferences.

The Adaptation Integration System [MPAA16] facilitates the application of accessibility adaptation rules at both design-time and run-time in a transparent way. The tool supports modifying the models at any level of CRF except the FUI (for which the rules are applied by transformations from the CUI). At design-time, GUI designers can use the Adaptation Integration System without prior knowledge about accessibility by providing it with an user interface description as well as a parameter describing user disabilities. The tool describes adaptation rules using AAL-DL[28] and uses MARIA [PSS09] as UIDL, but supports language transformations (e.g., from and to UsiXML). At run-time, the adaptation of the GUI is triggered by changes in the context of use, which are modeled using the Serenoa run-time context model [29] and detected by a context manager. The run-time adaptations facilitate, e.g., a dynamic change in the GUI modality based on context changes. For example, if a user changes from a silent to a noisy environment, the GUI exchanges audio elements with textual elements. The accessibility adaptation rules are provided by the Adaptation Integration System and cannot be modified or extended by model designers. Support for assistive technologies (e.g., activating them when needed) is missing.

[27]http://www.usixml.org/ (Last Access: 2017-05-21)

[28]https://pdfs.semanticscholar.org/4b05/afd0ebeba74de2c97d61b401e271b201943c.pdf (Last Access: 2017-05-25)

[29]https://pdfs.semanticscholar.org/bf9e/0295d38bc346164819be575e0453e809f1bd.pdf

SUPPLE [GLW06] leverages a run-time approach for automatically generating a custom user interface based on a Functional UI Model (i.e., an abstract description of the user interface) and a Custom Interaction Model. To generate the Custom Interaction Model, the user has to go through a one-time personalization process lasting up to 20 minutes, where he or she, e.g., compares pairs of concrete user interfaces. SUPPLE then uses decision-theoretic optimization to generate an optimized GUI from these two models. The personalized GUIs can differ in their widgets, layout and navigation structure. Its successor, SUPPLE++ [GWW07], generates customized GUIs optimized for users with motor impairments based on their actual (quantitative) performance in motor tasks instead of mere (qualitative) preferences. The generated GUI aims to minimize a user's expected movement time, but might also be optimized for users with visual impairments.

The MyUI [PHJS12] framework and infrastructure also follows a run-time approach and provides a publicly available and extensible multimodal design pattern repository for adapting a GUI based on context changes. The intention of this research project was to provide an extensible system to integrate all aspects of accessibility into the GUI generation process. Designers and developers can create new design patterns in the design pattern repository, which can be refined and reviewed by other accessibility experts. The design patterns include a human-readable description of the addressed problem and its solution as well as source code for the corresponding software component that implements the solution. Before the GUI can be adapted dynamically, a User Interface Profile is created. The profile is built from three parts: the Device Profile, the User Profile and the Customization Profile (which is defined by the developer). The User Interface Profile is created at the beginning of an interaction session with the GUI and then updated on any significant context change. Based on the User Interface Profile and an Abstract Application Interaction Model (AAIM), the most suitable GUI components for the current situation are selected. After this step, the final GUI is rendered. Adaptations of the GUI are system-initiated, i.e., relevant interaction events are recognized by a Context Manager which updates the profiles automatically and triggers the GUI adaptation process. To make the adaptations clear and understandable to the user, two adaptation patterns are used. On the one hand, adaptation rendering patterns use animations to seamlessly transform one GUI element into another (e.g., to switch from smaller fonts to larger fonts). Animations are used to draw the user's attention to the changing elements and to help him or her to understand how the elements changed. However, adaptation dialog patterns present the changes in a dialog which have to be confirmed by the user.

Yigitbas et al. [YSE17] developed an IDE for developing self-adapting UIs. They extended the common model-driven approach of having abstract models and domain models transformed into a FUI using a UI generator by two additional development paths which are responsible for UI adaptation and context management. In one development path, an abstract Adaptation Model is created, which is then transformed by the Adaptation Service Generator into an Adaptation Service that will adapt the FUI at run-time. The other path involves transforming a Context Model, which is referenced by the Adaptation Model, into a Context Service (e.g., accelerator, GPS). Yigitbas et al. defined three types of adaptations: Task-Feature-Set (e.g., showing or hiding certain components), Navigation (e.g., modifying links), and Layout (e.g., changing colors, font sizes or the way components are split among different screens). The resulting GUI is based on Angular 2.

Hussain et al. [HUHMB$^+$18] developed an adaptive UI/UX authoring tool that automatically evaluates a user's context and experience and adapts the UI accordingly. The context-of-use is based on three entities: user (e.g., preferences, abilities, goals), platform (e.g., resolution, connectivity), and environment (e.g., noise, light). It is observed implicitly (via user interaction) and explicitly (via questionnaires). At run-time, first a default UI is being displayed and the user has to register him- or herself. During the registration process, data for the models describing the context-of-use are collected. When the user then logs in to the application, a Context Evaluator adds additional information about the context, and the Adaptation Engine selects relevant rules and applies conflict resolution to generate an adapted UI. An evaluation with 32 participants showed that the consistency of the UI and thus the learning ability decreased if adaptations were applied recurrently.

Run-time approaches can provide benefits, e.g., that the GUI can be tailored to the specific needs of each user. However, GUI customizations and adaptations at run-time can take a substantial amount of time and may exceed acceptable performance times [PHJS12]. Also, current approaches regarding adaptive UIs typically are limited to specific domains and are not applicable to arbitrary scenarios [HUHMB$^+$18].

CHAPTER 3

Case Study on Accessibility in UCP

In order to determine the current status regarding accessibility of UCP, a case study was conducted, where six sample applications built with UCP have been analyzed. The evaluation of the accessibility of UCP has been done using multiple analysis techniques, including automated as well as manual approaches, to obtain a detailed view of the current status. Several accessibility shortcomings have been found, some of which are relevant to the process of generating GUIs, others are a result of out-dated implementation techniques.

The remainder of this chapter is organized as follows: Section 3.1 describes the reference applications built with UCP which were used for the accessibility analysis. The evaluation process that was used to analyze the status quo regarding accessibility in UCP is depicted in Section 3.2. In Section 3.3, the accessibility issues found during the evaluation are listed. Section 3.4 reflects on the lessons learned from the case study and elaborates potential solutions to the issues found.

3.1 The Reference Applications

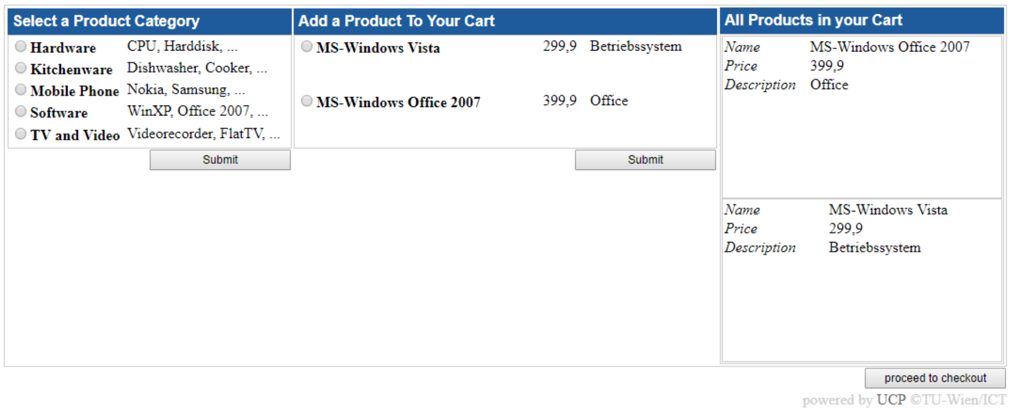
When a GUI generation framework has to be analyzed for its compliance to accessibility standards, one has to evaluate the compliance of reference applications generated by the framework. These reference applications have to cover the generation framework's capabilities. In case of UCP, such reference applications, provided by the developers of UCP, have already been publicly available at the time of writing this thesis. They have been created by the developers of UCP and are prototypes in a variety of business domains with differing levels of complexity, focusing on different features of UCP.

Seven applications were publicly available on the UCP Website[1] at the time of writing this thesis. Five of them have been used for this case study. The other two, Flight Booking Round Trip[2] and Hotel Booking[3], did not provide any further insight, since they were too similar to some of the other applications. Thus, they were excluded from the analysis. While evaluating the accessibility of the existing prototype applications, a use case was found in which a model designer could produce an application that violates accessibility standards by using *custom rules* [RKP15a]. To cover this use case in a sample application (called Flight Booking Round Trip Accessibility), the UCP developers provided a variation of the Flight Booking Round Trip prototype (extended by a *custom rule*), which was then also included in the analysis.

The seven applications analyzed in this case study are described below. Shopping, Travel Booking and Booking Kärnten were fully-automatically generated without any customization. In the Flight Booking prototype, a *custom widget* [RPK16] was included. The Accommodation Booking prototype, which is the most complex of all available prototypes, contains customized CSS as well as *custom rules*.

3.1.1 Shopping

The Shopping[4] application (see Figure 3.1 for the first screen displayed on a desktop computer) is a prototype of an online shop for computer hardware and other electronic devices. The user can select a product category, add a product to the cart, and buy the product(s) by filling out a form requesting the user's payment details, delivery and billing address. After submitting the form, the user receives a success message if everything went fine, otherwise an error message is displayed (e.g., if relevant data is missing in the form).


Select a Product Category

○ Hardware	CPU, Harddisk, ...
○ Kitchenware	Dishwasher, Cooker, ...
○ Mobile Phone	Nokia, Samsung, ...
○ Software	WinXP, Office 2007, ...
○ TV and Video	Videorecorder, FlatTV, ...

Submit

Add a Product To Your Cart

○ MS-Windows Vista	299,9	Betriebssystem
○ MS-Windows Office 2007	399,9	Office

Submit

All Products in your Cart

Name	MS-Windows Office 2007
Price	399,9
Description	Office

Name	MS-Windows Vista
Price	299,9
Description	Betriebssystem

proceed to checkout

powered by UCP ©TU-Wien/ICT

Figure 3.1: First screen of the Shopping prototype displayed on a desktop computer

[1]http://ucp.ict.tuwien.ac.at/ (Last Access: 2017-08-20)
[2]http://ucp.ict.tuwien.ac.at/UI/FlightBookingRoundTrip (Last Access: 2017-08-20)
[3]http://ucp.ict.tuwien.ac.at/UI/HotelBooking (Last Access: 2017-08-20)
[4]http://ucp.ict.tuwien.ac.at/UI/Shopping (Last Access: 2017-08-20)

The prototype was built using device specifications of a desktop computer and a mobile phone, respectively. Hence it displays a tailored GUI for small screens when accessing it via mobile phone (see Figure 3.2).

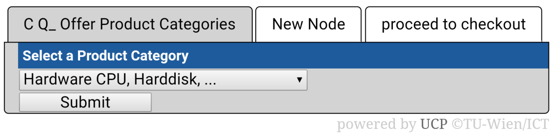
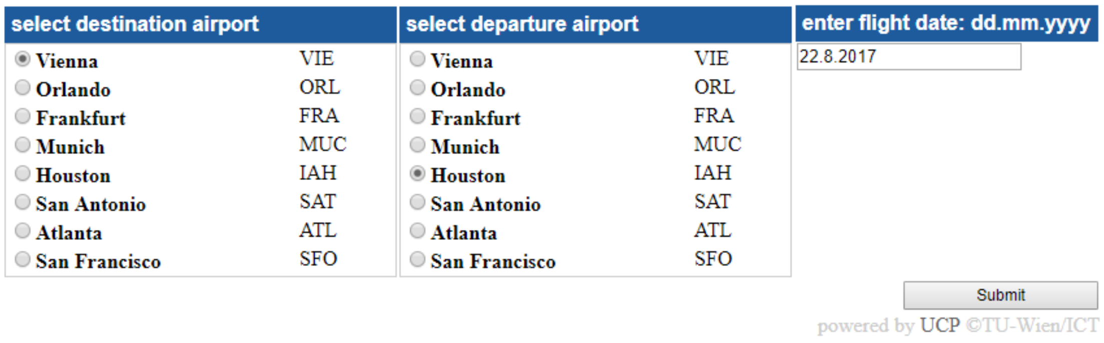

C Q_ Offer Product Categories
New Node
proceed to checkout
Select a Product Category
Hardware CPU, Harddisk, ...
Submit
powered by UCP ©TU-Wien/ICT

Figure 3.2: First screen of the Shopping prototype displayed on a mobile phone

3.1.2 Flight Booking

In the Flight Booking[5] prototype, shown in Figure 3.3, a user can book a flight by selecting a departure and destination airport and a flight date. The user is then requested to select a flight, followed by a graphical view in which the user can select an available seat. After entering credit card and passenger details, the user finally receives the purchased ticket number.



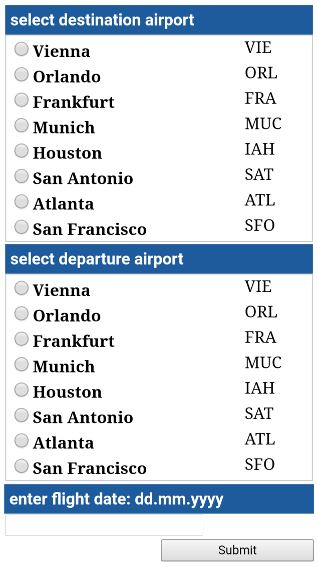
select destination airport		select departure airport		enter flight date: dd.mm.yyyy
● Vienna	VIE	○ Vienna	VIE	22.8.2017
○ Orlando	ORL	○ Orlando	ORL	
○ Frankfurt	FRA	○ Frankfurt	FRA	
○ Munich	MUC	○ Munich	MUC	
○ Houston	IAH	● Houston	IAH	
○ San Antonio	SAT	○ San Antonio	SAT	
○ Atlanta	ATL	○ Atlanta	ATL	
○ San Francisco	SFO	○ San Francisco	SFO	

Figure 3.3: First screen of the Flight Booking prototype displayed on a desktop computer

The prototype also contains a tailored GUI for mobile phones, which can be seen in Figure 3.4.

The prototype was customized by including a *custom widget* [RPK16] representing a seat picker (see Figure 3.5). It shows an illustration of the cabin of a plane, where already taken seats are displayed in black, and available seats are displayed in blue, which are selectable. A custom widget [RPK16] is an extension to the UCP framework's predefined GUI widgets and is included in the fully-automatic generation process. Custom widgets can be provided by a widget designer by creating a *Custom Widget Template* containing the widget's implementation along with some *Custom Widget Rules*, which map a specific part of the *Discourse Model* to the custom widget and thereby create a placeholder for it during generation. To enable design-time variability, custom widgets

[5]http://ucp.ict.tuwien.ac.at/UI/FlightBooking (Last Access: 2017-08-20)



select destination airport

Vienna	VIE
Orlando	ORL
Frankfurt	FRA
Munich	MUC
Houston	IAH
San Antonio	SAT
Atlanta	ATL
San Francisco	SFO

select departure airport

Vienna	VIE
Orlando	ORL
Frankfurt	FRA
Munich	MUC
Houston	IAH
San Antonio	SAT
Atlanta	ATL
San Francisco	SFO

enter flight date: dd.mm.yyyy

Figure 3.4: First screen of the Flight Booking prototype displayed on a mobile phone

can be parameterized using variables defined by the widget designer. GUI designers can then use these custom widgets without having in-depth knowledge of the UCP framework internals or editing the source code themselves.

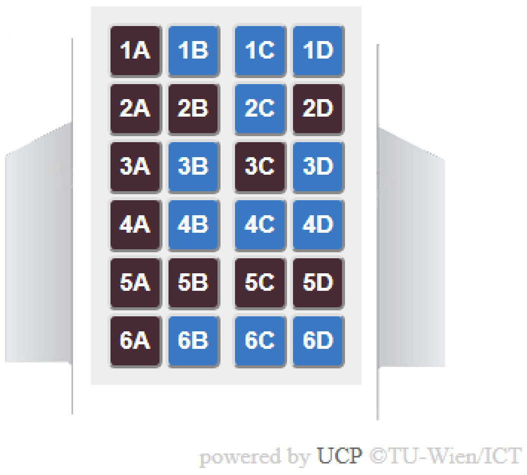
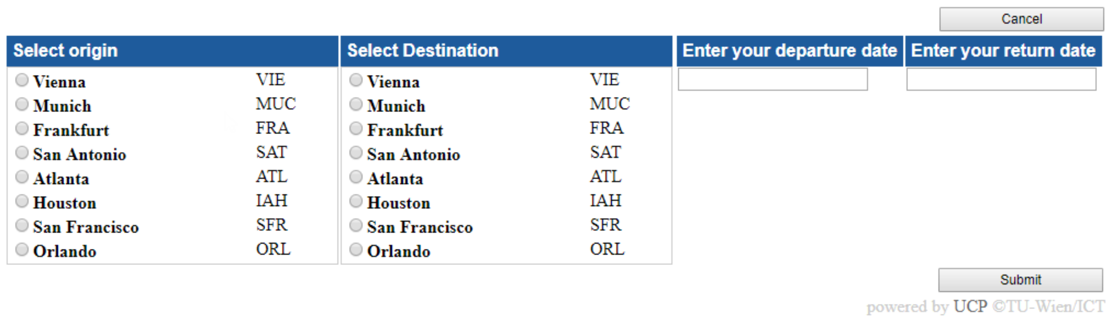


Figure 3.5: Seat picker custom widget of the Flight Booking prototype

3.1.3 Flight Booking Round Trip Accessibility

The Flight Booking Round Trip Accessibility[6] prototype, which can be seen in Figure 3.6, is very similar to the Flight Booking application, except for allowing the user to book a round-trip flight. This prototype was not built using multiple device specifications, hence it does not have a tailored GUI for small screens.



Cancel

Select origin		Select Destination		Enter your departure date	Enter your return date
○ **Vienna**	VIE	○ **Vienna**	VIE		
○ **Munich**	MUC	○ **Munich**	MUC		
○ **Frankfurt**	FRA	○ **Frankfurt**	FRA		
○ **San Antonio**	SAT	○ **San Antonio**	SAT		
○ **Atlanta**	ATL	○ **Atlanta**	ATL		
○ **Houston**	IAH	○ **Houston**	IAH		
○ **San Francisco**	SFR	○ **San Francisco**	SFR		
○ **Orlando**	ORL	○ **Orlando**	ORL		

Submit

powered by UCP ©TU-Wien/ICT

Figure 3.6: First screen of the Flight Booking Round Trip Accessibility prototype displayed on a desktop computer

Flight Booking Round Trip Accessibility was derived from the Flight Booking Round Trip prototype, which was not included in the accessibility evaluation, to demonstrate a special case of adaptation by the model designer using *custom rules* [RKP15a], in which a success criterion of a WCAG guideline was violated. This violation did not occur in the already existing prototypes, although it is a possible outcome of UCP's fully-automatic generation process (i.e., without manually adapting the application's source code afterwards).

The only change to the Flight Booking Round Trip prototype was that the "Cancel" button on the first screen was moved to the top right corner of the screen, which can be seen in Figure 3.6. In the original prototype, the "Cancel" button was placed next to the "Submit" button.

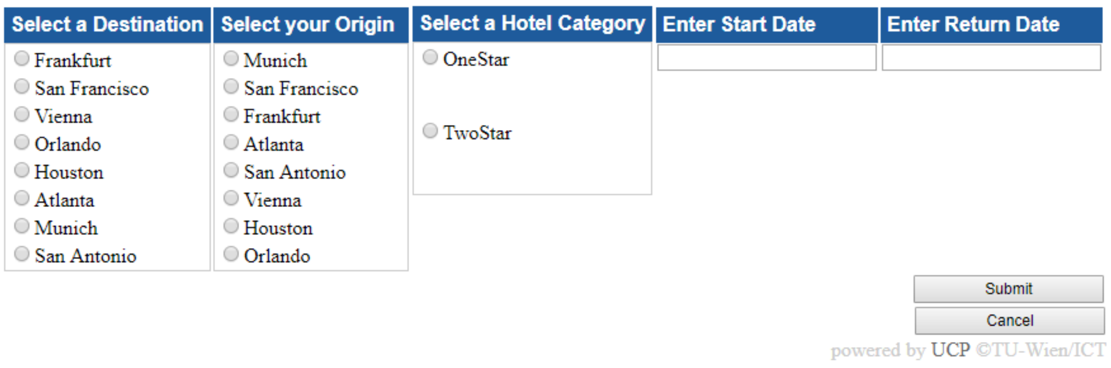
Custom rules [RKP15a] provide a way to customize a GUI through transformations during the automatic generation process. They can be specializations of UCP's basic rule set (i.e., they can be extensions of existing rules). While they can be created specifically for certain applications, there is also a way to provide so-called "generic-custom rules", which are predefined and do not depend on specific model structures in the *Discourse Model*. During generation, the transformation engine can match multiple rules for a specific discourse pattern, creating a search space of different potential GUIs for a single *Communication Model*. Which GUI is selected for generation is based upon constraints (e.g., available screen space). To enforce custom rules to be applied in the final GUI, the transformation engine inhibits existing basic rules matched by the same discourse patterns as a custom rule. While custom rules are specified manually, the final source code

[6] http://ucp.ict.tuwien.ac.at/UI/FlightBookingRoundTripAccessibility (Last Access: 2017-08-20)

is generated automatically. This enables the GUI designer to make the customizations persistent for re-generation and also to potentially re-use the same rules for multiple device specifications (as it was done in the Accommodation Booking prototype).

3.1.4 Travel Booking

In the Travel Booking[7] prototype (see Figure 3.7), a user can book a hotel in addition to a round-trip flight. After selecting an origin and a destination place, as well as a hotel category and the start and end date of the trip, the user can choose a flight and return flight as well as a specific hotel. Then, he chooses a room within the hotel. After that, he needs to fill out a form requesting the user's credit card details, payment details and personal details. Finally, the prototype displays a reservation number to the user. The application's GUI is not tailored to small devices.



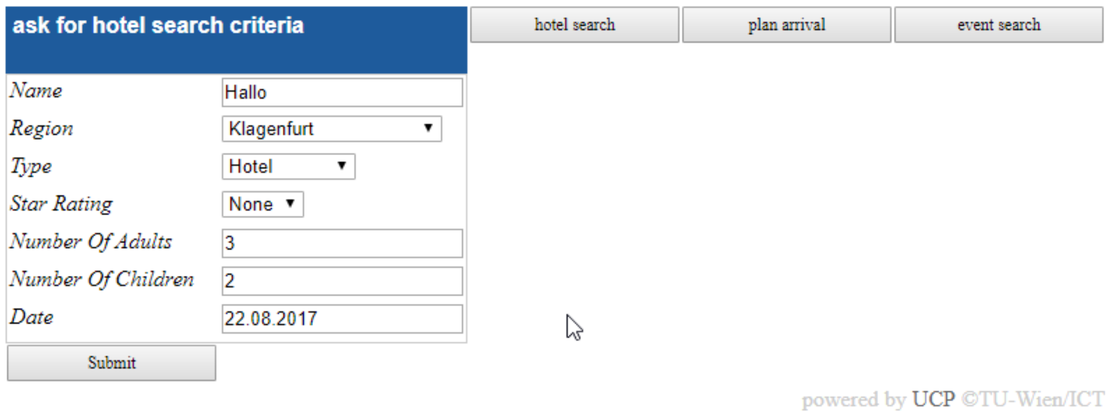
Select a Destination	Select your Origin	Select a Hotel Category	Enter Start Date	Enter Return Date
○ Frankfurt	○ Munich	○ OneStar		
○ San Francisco	○ San Francisco			
○ Vienna	○ Frankfurt	○ TwoStar		
○ Orlando	○ Atlanta			
○ Houston	○ San Antonio			
○ Atlanta	○ Vienna			
○ Munich	○ Houston			
○ San Antonio	○ Orlando			

Figure 3.7: First screen of the Travel Booking prototype displayed on a desktop computer

3.1.5 Booking Kärnten

The Booking Kärnten[8] prototype lets a user plan a trip to the Austrian region Kärnten including searching for matching transportation, hotels and events (see Figure 3.8). It includes a navigation for switching between hotel search, arrival planning and event search. For the hotel search, the user can provide more detailed information than in the Travel booking prototype, and more details are shown for the respective hotels. The user can also search for events by date, description and region and purchase a train ticket to the destination. The application is not tailored to small devices.

[7]http://ucp.ict.tuwien.ac.at/UI/travelBooking (Last Access: 2017-08-20)
[8]http://ucp.ict.tuwien.ac.at/UI/BookingKaernten (Last Access: 2017-08-20)


ask for hotel search criteria

Name	Hallo
Region	Klagenfurt
Type	Hotel
Star Rating	None
Number Of Adults	3
Number Of Children	2
Date	22.08.2017

Figure 3.8: First screen of the Booking Kärnten prototype displayed on a desktop computer

3.1.6 Accommodation Booking

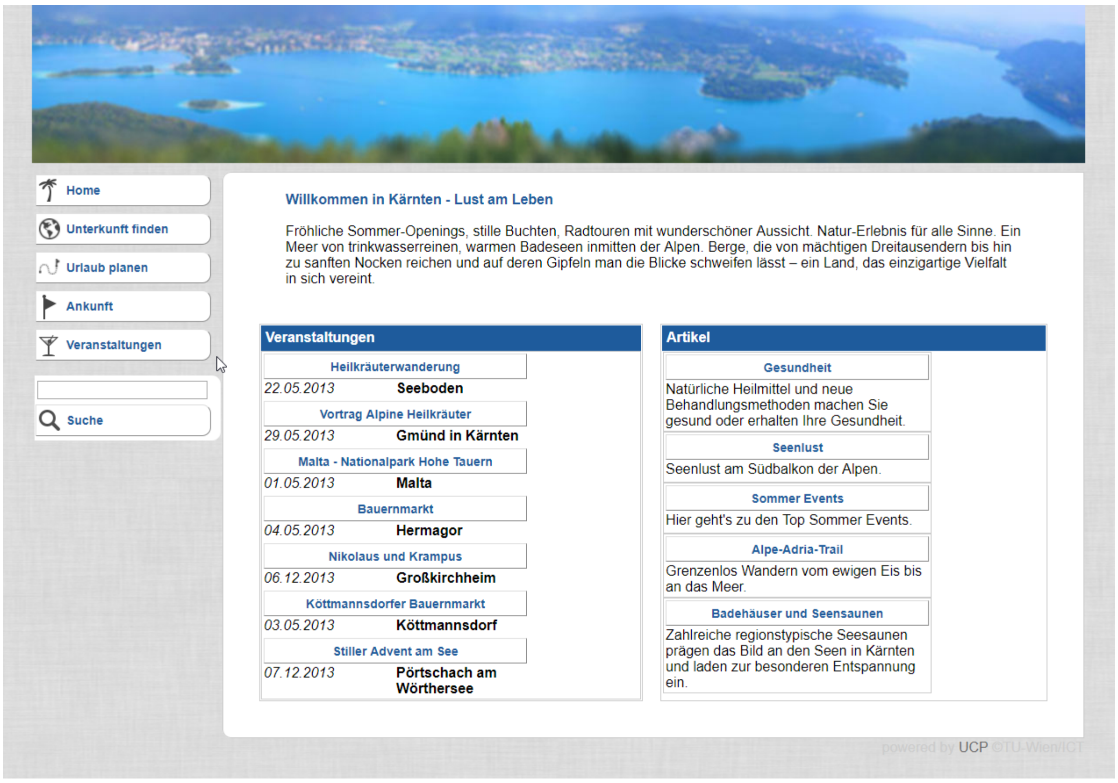
The Accommodation Booking[9] application, which can be seen in Figure 3.9, is the most complex prototype of the reference applications. It is an extension of the BookingKärnten prototype and was built using additional custom rules, custom CSS and layout hints. It is also tailored to small devices, as can be seen in Figure 3.10.

The Accommodation Booking prototype has the look-and-feel of a typical Website, including a header image and a navigation on the left side of the screen (on desktop computers). The navigation consists of a search field for quickly finding accommodations, articles or events. The Homepage shows an overview of upcoming events and existing articles. The user can search for an accommodation and receives a list of matching results, each of them showing an overview of the respective hotel. For each result, there is a detail page, containing further information about the hotel and a form to request a booking. When choosing “Urlaub planen” (German for “plan holiday”) from the navigation, the user can specify the beginning and ending of the holiday, a preferred region and preferred activities, and receives a list of matching hotels. The user can find information for arriving via plane, train or car. Finally, the user can search for events and read information about them.

The tailored GUI for small devices looks different from the desktop GUI (see Figure 3.10). It is organized into a tab view containing two tabs, one for the current Web page and one for the navigation.

The Accommodation Booking prototype was generated using several custom rules, for example showing only a subset of attributes for a given object (e.g., a hotel information domain object from the DoD model) in a certain context (e.g., the search result list vs. the hotel’s detail page). Other examples show additional labels for clarification and the

[9]http://ucp.ict.tuwien.ac.at/UI/accomodationBooking (Last Access: 2017-08-20)


Home

Unterkunft finden

Urlaub planen

Ankunft

Veranstaltungen

Suche

Willkommen in Kärnten - Lust am Leben

Fröhliche Sommer-Openings, stille Buchten, Radtouren mit wunderschöner Aussicht. Natur-Erlebnis für alle Sinne. Ein Meer von trinkwasserreinen, warmen Badeseen inmitten der Alpen. Berge, die von mächtigen Dreitausendern bis hin zu sanften Nocken reichen und auf deren Gipfeln man die Blicke schweifen lässt – ein Land, das einzigartige Vielfalt in sich vereint.

Veranstaltungen

Heilkräuterwanderung	
22.05.2013	Seeboden
Vortrag Alpine Heilkräuter	
29.05.2013	Gmünd in Kärnten
Malta - Nationalpark Hohe Tauern	
01.05.2013	Malta
Bauernmarkt	
04.05.2013	Hermagor
Nikolaus und Krampus	
06.12.2013	Großkirchheim
Köttmannsdorfer Bauernmarkt	
03.05.2013	Köttmannsdorf
Stiller Advent am See	
07.12.2013	Pörtschach am Wörthersee

Artikel

Gesundheit

Natürliche Heilmittel und neue Behandlungsmethoden machen Sie gesund oder erhalten Ihre Gesundheit.

Seenlust

Seenlust am Südbalkon der Alpen.

Sommer Events

Hier geht's zu den Top Sommer Events.

Alpe-Adria-Trail

Grenzenlos Wandern vom ewigen Eis bis an das Meer.

Badehäuser und Seensaunen

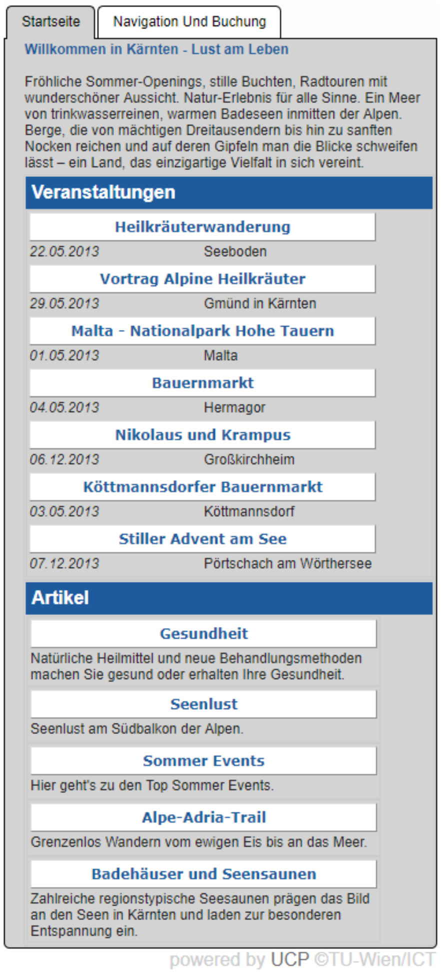
Zahlreiche regionstypische Seesaunen prägen das Bild an den Seen in Kärnten und laden zur besonderen Entspannung ein.

powered by UCP ©TU-Wien/ICT

Figure 3.9: First screen of the Accommodation Booking prototype displayed on a desktop computer

number of search results at the top of the page. Some of these rules could be re-used for multiple device specifications. In case the rules could not be re-used (e.g., because the size of the resulting widgets was too large for a small device), the existing custom rules could be adapted to fit the respective device.

In cases where specific layout rules needed to be applied, so-called *layout hints* [RPV12] were used. Layout hints explicitly specify certain layout parameters via transformation rules. They were introduced because a widget's position is strongly influenced by its functionality, which is not generic. For example, one would expect buttons used for navigation (e.g., a "Home" button to get back to the first screen of the application) to be at the top of the screen, while a button used for submitting a form is typically expected to be at the end of the form. Layout hints can be used in these cases to align elements within their container. Other options are, for example, to specify the widget ordering strategy (e.g., place the biggest widget first and then fill the rest of the space with smaller widgets) or the widget insertion strategy (smallest waste space versus best ratio). In the Accommodation Booking prototype, layout hints were used to place the navigation at the left of the screen on desktop computers and to define the ordering of the navigation entries. Also, the "Back" buttons were placed at the top right of the page.



Startseite | Navigation Und Buchung

Willkommen in Kärnten - Lust am Leben

Fröhliche Sommer-Openings, stille Buchten, Radtouren mit wunderschöner Aussicht. Natur-Erlebnis für alle Sinne. Ein Meer von trinkwasserreinen, warmen Badeseen inmitten der Alpen. Berge, die von mächtigen Dreitausendern bis hin zu sanften Nocken reichen und auf deren Gipfeln man die Blicke schweifen lässt – ein Land, das einzigartige Vielfalt in sich vereint.

Veranstaltungen

Heilkräuterwanderung
22.05.2013 Seeboden

Vortrag Alpine Heilkräuter
29.05.2013 Gmünd in Kärnten

Malta - Nationalpark Hohe Tauern
01.05.2013 Malta

Bauernmarkt
04.05.2013 Hermagor

Nikolaus und Krampus
06.12.2013 Großkirchheim

Köttmannsdorfer Bauernmarkt
03.05.2013 Köttmannsdorf

Stiller Advent am See
07.12.2013 Pörtschach am Wörthersee

Artikel

Gesundheit
Natürliche Heilmittel und neue Behandlungsmethoden machen Sie gesund oder erhalten Ihre Gesundheit.

Seenlust
Seenlust am Südbalkon der Alpen.

Sommer Events
Hier geht's zu den Top Sommer Events.

Alpe-Adria-Trail
Grenzenlos Wandern vom ewigen Eis bis an das Meer.

Badehäuser und Seensaunen
Zahlreiche regionstypische Seesaunen prägen das Bild an den Seen in Kärnten und laden zur besonderen Entspannung ein.

powered by UCP ©TU-Wien/ICT

Figure 3.10: First screen of the Accommodation Booking prototype displayed on a mobile phone

3.2 The Evaluation Process

For the accessibility evaluation process, the WCAG-EM was employed. This was due to the fact that the purpose of the evaluation was to identify whether UCP adheres to legally required accessibility guidelines, which cover a wide variety of disabilities. Since the methodology provided by Orozco et al. [OTD16] focuses on addressing very specific types of disabilities, it was not very suitable for this evaluation step.

Altogether, two evaluation cycles have been performed. During the first cycle performed in February 2017, the prototype applications were evaluated against the conformance level AA of the WCAG 2.0 guidelines. This conformance level was chosen since the EN 301 549 matches it, which is the minimum requirement for Web and mobile accessibility defined by the most recent European directive on ICT accessibility (see Section 2.2.2).

The second cycle, performed in July 2017, was carried out to evaluate against the recently published WCAG 2.1 guidelines (in the Working Draft of June 30, 2017), which were not yet available during the first evaluation run. The chosen target conformance level was raised to AAA in order to investigate the degree to which UCP could adhere to this highest level of conformance.

The reason for selecting the WCAG as the target standard to evaluate against was that the other standards relevant in Austria, ÖNORM EN ISO 9241-171 and EN 301 549, are both very similar to the WCAG guidelines and partly even derived from them. In contrast to the other two standards, the WCAG guidelines are also used world-wide and serve as a basis for multiple other standards in different countries. Since the current implementation of UCP uses only Web technologies, the WCAG guidelines seemed most appropriate. The guidelines were originally tailored to these technologies, but now are generalized to basically any technology (although the recommended techniques to meet the guidelines are still focused on Web technologies).

In both evaluation cycles, the same evaluation techniques (fully-automatic tools, manual inspection, screen reader tests), which are described and compared in Section 3.2.1, have been employed. The evaluation was carried out on a desktop computer (running Ubuntu Linux and Windows 10, respectively, with a screen resolution of 1920x1080 pixels) as well as a smartphone running Android 6 (with a screen resolution of 375x810 pixels). A mouse, a keyboard and two screen readers (JAWS 17 and Google Talkback, see Section 3.2.3) were used as input devices. The browsers used for the evaluation were:

- Chrome 55 (desktop and mobile version)
- Firefox 51
- Internet Explorer 11
- Edge 38

The second step of WCAG-EM involved exploring the target applications to gain an understanding of their use cases, functionalities and underlying technologies. UCP is based on HTML, CSS, JavaScript and the server-side Java template engine Velocity. Since UCP currently is not capable of serving audio or video content, or timing-dependent functionalities (like e.g., session expiration or auto-updating content), using its basic functionalities, some guidelines from WCAG are not relevant to the prototypes. These guidelines are:

- Guideline 1.1 - Time-based Media
- Guideline 1.4.2 - Audio Control
- Guideline 1.4.7 - Low or No Background Audio
- Guideline 2.2 - Enough Time

- Guideline 2.3 - Seizures

The third step according to WCAG-EM was the selection of a representative sample of the evaluated applications. Since the scope of the UCP prototype applications is manageable, all screens were taken into account for the evaluation and hence this step was skipped.

After defining the evaluation scope, the UCP prototype applications have been audited. Several evaluation techniques have been employed (see Section 3.2.1). A report of the discovered accessibility issues of the UCP framework is given in Section 3.3.

3.2.1 Applied Evaluation Techniques

During the two evaluation cycles, each screen (and also each complete use case) of each of the UCP prototype applications was evaluated in detail and reviewed whether it adheres to the conformance requirements. A combination of multiple evaluation techniques has been applied. To get a rough overview of the current accessibility conformance status of the applications and their most salient issues, several *automated evaluation tools* and *automated validators*, which are listed in Section 3.2.3, have been employed. The results from these tools had to be validated manually.

After this step, a *manual inspection* was performed, for which the WCAG 2.0 and 2.1 guidelines and documents (see Section 3.2.2) have been utilized. For every relevant success criterion of the respective conformance level from the WCAG guidelines, each screen and use case of each application was reviewed manually. This was done on the desktop computer as well as on the smartphone.

Finally, a *screening technique* (or simulation kit approach) was applied by using a screen reader (while turning off the monitor) and a keyboard (or the touchscreen on the smartphone, respectively) to navigate through the applications.

All in all, three out of four different methods from the classification of Bai et al. [BMSF16] (see Section 2.5) were used: the automatic/semi-automatic approach using automated evaluation tools and validators, the expert testing approach (i.e., performing a manual inspection using the WCAG guidelines), and the simulation kit approach (by using a screen reader while turning off the monitor, and using only the keyboard/touchscreen as input device).

The decision to choose these evaluation techniques was based on various studies ([Bra08], [BYH10], [MFT05], [VBC13]) regarding the efficiency of different methods. Brajnik [Bra08] compared the advantages and disadvantages of multiple techniques.

The conformance review, also called expert review or manual inspection, is the most widely used technique and can be used for formative and summative evaluations [Bra08]. Its great advantage is the ability to detect a wide range of different accessibility shortcomings for a variety of disabilities. Checking an application's conformance to a certain set of guidelines is also a very cost-efficient evaluation technique. By using guidelines derived from regulative requirements, the evaluator can also make sure that the application adheres to applicable laws. However, the quality of an evaluation also strongly depends on the guidelines being used. If the guidelines do not provide detailed and reproducible evaluation process descriptions as well as sensible success and failure criteria, the evaluator will likely miss accessibility issues that disabled users might actually face in reality.

Automated tools are not viable to be used as the only technique for an accessibility evaluation, as already discussed in Section 2.5.2. Nevertheless, because of their systematic and automated nature, they can increase productivity and can easily cover a wide range of Web pages in an application [Bra08]. When used carefully and validating their results manually, fully-automated accessibility evaluation tools can become a suitable enhancement of the evaluation process, especially for large applications. Vigo et al. [VBC13] investigated the effectiveness of six fully-automated tools with respect to WCAG 2.0 conformance checking. They claim that since these tools cannot interpret context, they fail to find certain accessibility issues on the one hand, while reporting false positives (i.e., supposed issues which are not actual issues) on the other hand. Coverage (the number of success criteria reported at least once) was generally quite low (50 % or less) for all tools. Success criteria from the Perceivable and Operable guidelines of WCAG 2.0 were covered the least. Completeness, i.e., the ratio between the number of reported and actual violations, was best for the Robust guideline (larger than 73 %), while the performance for the Operable and Understandable guidelines was much lower (between 14 and 47 %). Tools which scored higher with regard to completeness usually had lower correctness scores, i.e., catching more violations resulted in producing more false positives, too. The overall accessibility of the evaluated Websites had an effect on the tools, too: the more inaccessible a Website was, the higher the completeness scores were. According to Vigo et al., automatic evaluation tools are better in finding more frequent accessibility issues, especially regarding the Perceivable guideline, than finding more subtle issues. They also performed better in finding more crucial issues, i.e., those which belong to the conformance level A, than finding issues of lower importance (i.e., conformance level AA and above) [VBC13]. These results suggest that fully-automatic tools are suitable to use in the beginning of an evaluation to discover the most salient issues and gain a rough overview of the accessibility of an application. Hence, fully-automatic tools were used in the beginning of the evaluation of the UCP prototypes.

Screening techniques, i.e., artificial degradation of sensory, motor or cognitive capabilities while using the application under evaluation (like the use of a screen reader instead of a monitor), are suitable for finding normative as well as empirical issues concerning certain kinds of disabilities [Bra08]. On the other hand, this technique does not apply a systematic approach, hence it might fail to find all issues. It also would not be practicable when being the only one technique applied, since many different kinds of screening techniques would have to be used in order to cover a wide variety of issues. When it comes to the effectiveness of screening techniques, the results found in research studies are inconclusive. While Brajnik [Bra08] claims that they are highly dependent on the experience of the evaluators with the assistive technologies, Mankoff et al. [MFT05] found that multiple developers using screen readers performed best compared to laboratory studies with blind users, automated tools, developers without screen readers and remote testing with blind users. Using screen readers also reduced the number of false positives compared to evaluations without screen readers [MFT05]. Since the screening technique using screen readers showed success in this study, it was chosen to be used also for the evaluation of UCP in addition to the other techniques.

Testing with users can be done in several ways, one of which is the think-aloud protocol, where a user speaks out loud what he thinks while navigating through the application. A benefit of testing with users is, of course, that issues are found which concern real users and real usage scenarios. In contrast, this technique might fail to find relevant issues if the user group is not chosen well (i.e., it represents most of the common disabilities and varying experience levels). Additionally, the results might mirror not only accessibility problems, but also usability issues that affect all users in general. Mankoff et al. [MFT05] also found that testing with users (at least if performed remotely) was among the worst performing evaluation methods when evaluating the accessibility of Web pages for blind users. A reason for this might be that the users were too experienced in screen reader usage and hence failed to find issues that users with lower experience commonly have. Since this thesis addresses accessibility issues and not usability in general, and because finding a representative user group was not feasible, testing with users was not part of the accessibility evaluation of UCP.

3.2.2 Used Guidelines and Documents

Since the prototype applications were evaluated against the WCAG guidelines during the first evaluation iteration, the relevant documents addressing this standard have been taken into account. W3C provides the following documents, which have been used for the evaluation:

- *WCAG 2.0*[10] *and 2.1*[11] *Guidelines:* These are the main documents of the WCAG guidelines containing the actual standard including the success criteria to be checked for each guideline.

- *How to Meet WCAG 2.0*[12] *and 2.1*[13]*:* These documents are basically checklists containing a list of links to techniques and failures for each guideline and its success criteria. They also link to the according pages of the *Understanding WCAG 2.0 and 2.1* documents and provide filters for certain tags, conformance levels, techniques, failures and technologies.

[10]https://www.w3.org/TR/WCAG20/ (Last Access: 2017-09-06)
[11]https://www.w3.org/TR/WCAG21/ (Last Access: 2017-09-06)
[12]https://www.w3.org/WAI/WCAG20/quickref/ (Last Access: 2017-09-06)
[13]https://www.w3.org/WAI/WCAG21/quickref/ (Last Access: 2017-09-06)

- *Understanding WCAG 2.0*[14] *and 2.1*[15]*:* These documents provide additional information to help developers and evaluators better understand why a certain guideline exists. For each guideline, its intent and advisory techniques are explained. Further, for each success criterion, its intents and benefits are clarified and some examples and related resources are given. Additionally, techniques to meet the success criterion and common failures regarding it are listed.

- *Techniques for WCAG 2.0*[16] *and 2.1*[17]*:* These documents provide details about sufficient techniques, advisory techniques and common failures for each of the guidelines and their success criteria. For each technique and failure, examples are shown and a description of test procedures for them is provided.

- *WCAG2ICT*[18]*:* This document explains how the WCAG 2.0 guidelines can be applied to non-Web software. Although UCP currently only generates Web-based software, it could potentially also use GUI toolkits for different technologies (e.g., the Java Swing[19] GUI toolkit, as UCP did in the past.

- *Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile*[20]*:* This document describes how the WCAG 2.0 guidelines can be applied to mobile Web applications.

During the second iteration of the evaluation, the *WCAG 2.1 Working Draft as of June 30 2017*[21] has been used. In this working draft, two new guidelines and nine new success criteria have been introduced.

3.2.3 Used Accessibility Evaluation Tools

Various tools have been used during the accessibility evaluation. In a first step, automated evaluation tools were used to discover some basic issues. HTML and CSS validators have been used to identify technical flaws in the source code. During manual evaluation, the prototype applications were navigated not only via keyboard and mouse, but also through screen reader and keyboard.

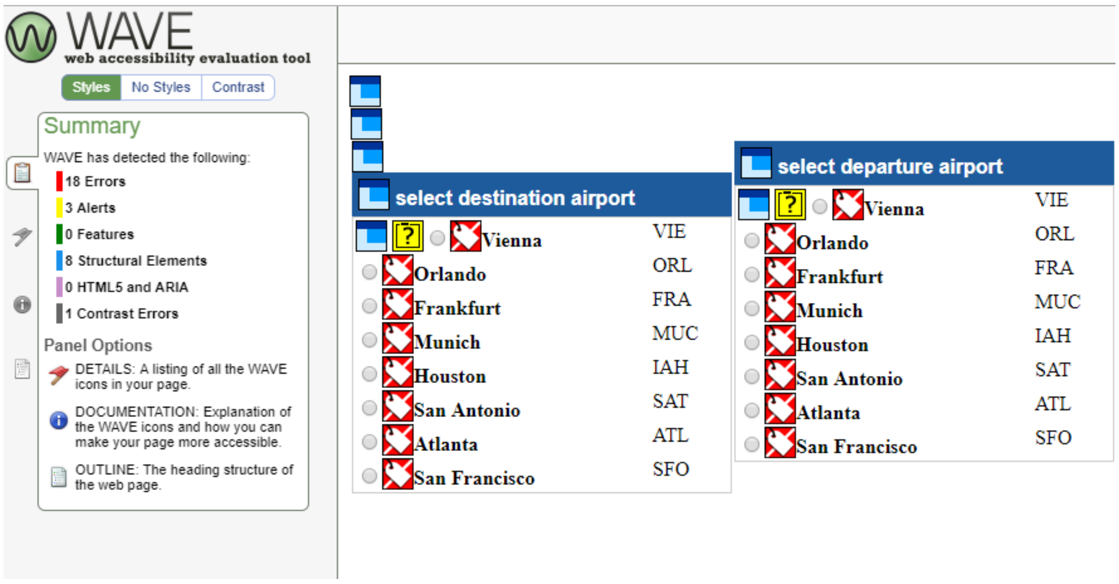
Automated Tools

To get a basic assessment of the accessibility of UCP, four fully-automated analysis tools were used in a first evaluation step.

[14]https://www.w3.org/TR/UNDERSTANDING-WCAG20/ (Last Access: 2017-09-06)
[15]https://www.w3.org/WAI/WCAG21/Understanding/ (Last Access: 2017-09-06)
[16]https://www.w3.org/TR/WCAG20-TECHS/ (Last Access: 2017-09-06)
[17]https://www.w3.org/WAI/WCAG21/Techniques/ (Last Access: 2017-09-06)
[18]https://www.w3.org/TR/wcag2ict/ (Last Access: 2017-09-07)
[19]https://docs.oracle.com/javase/tutorial/uiswing/index.html (Last Access: 2017-09-07)
[20]https://www.w3.org/TR/mobile-accessibility-mapping/ (Last Access: 2017-09-07)
[21]https://www.w3.org/TR/2017/WD-WCAG21-20170630/ (Last Access: 2017-09-07)

WAVE[22] is a widely used accessibility evaluation tool provided by WebAIM[23]. It is available online, and there are free browser extensions for Chrome and Firefox. The advantage of WAVE is that it can be conveniently used for applications which serve multiple screens under the same URL, like the UCP prototypes do (e.g., all screens of the Flight Booking prototype are served under http://ucp.ict.tuwien.ac.at/UI/FlightBooking), since the browser extension analyzes every currently viewed screen. Another specialty compared to other fully-automatic evaluation tools is that it provides a visual representation of the accessibility violations within the Web page using annotations and suggests ways to fix them.

Figure 3.11 illustrates such annotations in an example application generated by UCP. 18 errors (visualized by the red icons) have been found which relate to the fact that radio buttons do not have a semantic label (using the `<label>` tag) associated to them. The alerts (visualized as yellow icons) indicate that the radio buttons are also not enclosed by a fieldset. The blue icons indicate structural elements on the page (in this case layout tables). In the menu on the left side, detailed descriptions of the errors can be found.



WAVE has detected the following:

- 18 Errors
- 3 Alerts
- 0 Features
- 8 Structural Elements
- 0 HTML5 and ARIA
- 1 Contrast Errors

Panel Options

DETAILS: A listing of all the WAVE icons in your page.

DOCUMENTATION: Explanation of the WAVE icons and how you can make your page more accessible.

OUTLINE: The heading structure of the web page.

select destination airport	
Vienna	VIE
Orlando	ORL
Frankfurt	FRA
Munich	MUC
Houston	IAH
San Antonio	SAT
Atlanta	ATL
San Francisco	SFO

select departure airport	
Vienna	VIE
Orlando	ORL
Frankfurt	FRA
Munich	MUC
Houston	IAH
San Antonio	SAT
Atlanta	ATL
San Francisco	SFO

Figure 3.11: WAVE's Visual Representation of Accessibility Violations in the Flight Booking Prototype

SortSite Trial[24] is a free trial version of SortSite, a Website quality testing tool focusing on accessibility, Web standards and usability. It can be used online or downloaded for Windows and Mac OS X, for which it is usable for free for 30 days. Its aim is to analyze a complete Website by traversing all links from the same domain on a Web page recursively, using Web crawling techniques and scanning their contents for potential quality issues.

[22]http://wave.webaim.org/ (Last Access: 2017-09-07)
[23]http://webaim.org/ (Last Access: 2017-09-15)
[24]https://www.powermapper.com/products/sortsite/ (Last Access: 2017-09-07)

While this is certainly a convenient feature in most cases, for the UCP prototype applications it was a bit counter-productive. Since every prototype contained a link to the UCP project Website which was hosted under the same domain as the prototypes, SortSite traversed the project Website along with each prototype. Additionally, all screens except the first screen of the prototype applications were accessible using form buttons within the first screen (i.e., they were not linked to the first screen via hyperlinks), therefore SortSite could only analyze the first screen of each prototype. Hence, only a small subset of the analysis result was relevant for the accessibility evaluation of the prototypes, while the whole scanning procedure took clearly more time than for the other fully-automatic tools.

Total Validator[25] performs WCAG 2.0 compliance checking, along with HTML and CSS validation, spell checking and broken link checking. Its free Basic version is available for Windows, Mac OS X and Linux. The Basic version provides all accessibility validation features that the commercial Pro version does, but it does not automatically traverse a complete Website and is not able to analyze login protected parts of a Website (which the prototype applications did not have, anyway). There are also free browser extensions for Chrome and Firefox, which require Total Validator to be installed on the computer. It is not able to analyze multiple screens served under the same URL, hence, like SortSite, it could only analyze the first screen of each prototype.

TAW[26] is freely available as a Web tool and as a desktop application which provides more functionality than the Web tool. There is also a Firefox extension. For the evaluation, the desktop application was used. It categorizes issues found into certain violations (e.g., missing form input labels) and issues that require human judgment (e.g., meaningful alternative texts for images), which makes it easier to validate them. TAW also provides visual highlighting of issues within a local copy of a Web page (similarly to WAVE).

Since WCAG 2.1 did not yet reach the status of a W3C Recommendation, none of these tools supports newly introduced success criteria from WCAG 2.1. All four tools are officially listed on the Web Accessibility Evaluation Tools List[27] provided by the W3C.

The decision to use these tools was based on a study performed by Vigo et al. [VBC13], who benchmarked the effectiveness regarding the evaluation of WCAG 2.0 compliance of six state-of-the-art fully-automated accessibility evaluation tools. Among those six tools, TAW performed best with regard to coverage (i.e., the number of different success criteria correctly identified as violated) and completeness (i.e., the ratio of reported violations over actual violations). SortSite performed very well regarding correctness, which was defined as the ability to minimize the number of false positives (i.e., reported violations which are not actual violations), while TAW and TotalValidator had a much lower correctness (likely due to the higher completeness scores).

[25]https://www.totalvalidator.com/ (Last Access: 2017-09-07)

[26]http://www.tawdis.net/ingles.html?lang=en (Last Access: 2017-09-07)

[27]https://www.w3.org/WAI/ER/tools/ (Last Access: 2017-09-25)

TAW, TotalValidator and SortSite have been chosen for the evaluation because they performed well across all WCAG 2.0 principles. A combination of relatively high completeness and coverage (TAW and TotalValidator) and high correctness (SortSite) seemed like a balanced choice. Each of the automatically found violations was reviewed manually to exclude false positives. WAVE was not benchmarked by Vigo et al. [VBC13]. Yet, since it is a widely-used tool supporting easy analysis of multiple screens served from the same URL path via a browser extension, it was also included in the evaluation.

Validators

Both *HTML*[28] *and CSS*[29] *validators* have been applied on each application prototype. These validation services are provided by the W3C and support validation by URI (suitable for the first screen of a prototype) as well as validation by file upload or direct input (used for the remaining screens of a prototype).

Screen Readers

Two screen readers were used, one for desktop computers and one for mobile phones. The decision upon which screen readers to pick for the evaluation was guided by the *6th Screen Reader User Survey*[30] conducted by WebAIM in August 2015. A sample group of 2515 screen reader users, consisting not only of blind users, but also of people with low vision and cognitive disabilities, participated in that survey.

For the evaluation on a desktop computer, *JAWS 17*[31] has been used. JAWS was still the most widely used desktop computer screen reader among the sample group in 2015, although other screen reader solutions are becoming increasingly more popular recently. JAWS is available in seven languages and also supports output on braille displays. During the evaluation, the trial version of JAWS 17 has been used, which provides the full feature set, but is limited to a usage time of 40 minutes at a time, after which a restart of the computer is required to be able to use it again.

On mobile devices, *VoiceOver*[32] (a screen reader for iOS devices) was the most frequently used screen reader among the sample group, followed by *Google Talkback*[33] for Android. Because no iOS device was available for the first cycle of the evaluation, only Google TalkBack was used on a smartphone in this cycle. For the second evaluation cycle, Google Talkback was used on a smartphone and VoiceOver was used on a tablet. Google Talkback and VoiceOver are both standard features of Android and iOS systems, respectively, and can be activated in the accessibility settings.

[28] https://validator.w3.org (Last Access: 2017-09-07)

[29] https://jigsaw.w3.org/css-validator/ (Last Access: 2017-09-07)

[30] http://webaim.org/projects/screenreadersurvey6/ (Last Access: 2017-09-15)

[31] http://www.freedomscientific.com/Products/Blindness/JAWS (Last Access: 2017-09-07)

[32] https://www.apple.com/accessibility/iphone/vision/ (Last Access: 2017-09-07)

[33] https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback (Last Access: 2017-09-07)

3.3 Accessibility Issues Identified

Several accessibility issues were identified during the evaluation. About half of them concern criteria that are classified as level A of WCAG, and seven of them failed to meet level AA criteria. Issues were found for each of the four WCAG sections (Perceivable, Operable, Understandable, Robust), most problems (i.e., twelve) addressed the "Perceivable" section, followed by "Operable" (eight issues) and "Understandable" (seven issues). Four issues concern criteria from the updated WCAG 2.1 guidelines, most of them are problems regarding the appearance on mobile devices, which previously were only covered by the mobile accessibility mapping of WCAG 2.0[34]. Figure 3.12 depicts the distribution of the discovered shortcomings among the WCAG sections, and conformance levels grouped by sections, respectively. Five of the accessibility issues identified appear only on mobile devices. In the following, the shortcomings are explained in detail.

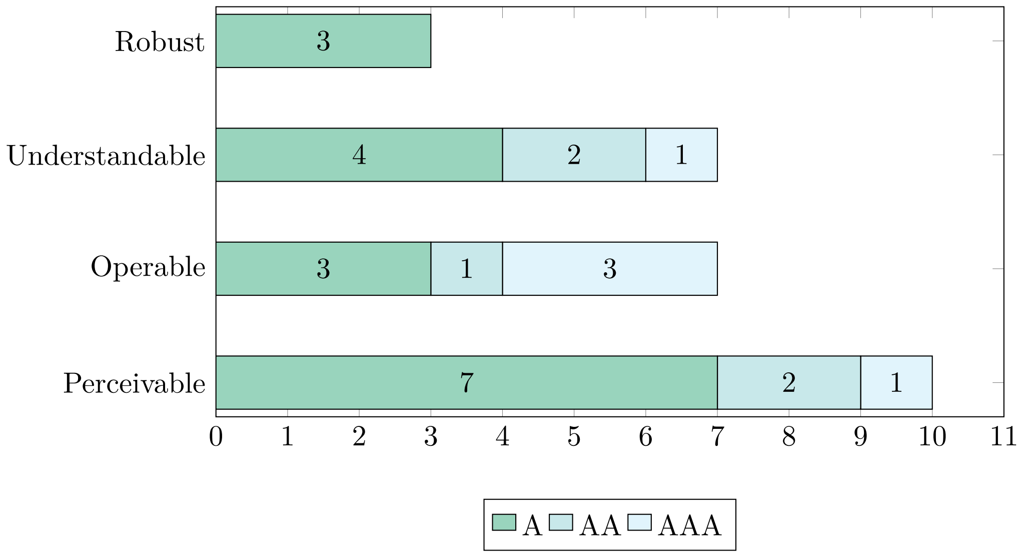


Figure 3.12: Distribution of discovered accessibility issues among the WCAG 2.1 sections

3.3.1 Accessibility Issues Failing Level A

Level A is the lowest conformance level of WCAG and required by the E-GovG. It covers the most essential criteria applicable to any domain, which can be applied with reasonable effort and do not limit the "look and feel" of the application [CCGV08].

Missing Semantic Information

Many discovered issues of the generated applications correspond to missing semantic information in the source code, which results in the inability of assistive technologies to access UI components or create perceivable and understandable output. These issues

[34]https://www.w3.org/TR/mobile-accessibility-mapping/ (Last Access: 2018-05-10)

were introduced because accessibility was not taken into consideration when UCP was created. Many developers do not know (or have not enough knowledge about) the WCAG and hence simply do not know which information is relevant for assistive technology.

To begin with, the generated applications' language cannot be determined programmatically. This is especially relevant for screen readers as they can utter text in the correct language based on this information, otherwise they would fall back to their default language, which may not match the application's language. In the Accommodation Booking application, the alternative texts of images are based on the respective image's filename, which is not descriptive at all, and even the language does not match the application's language (e.g., "Bild" for an image named `<UUID>__0__Bild.jpeg`). This makes it impossible for some users that cannot see the image to know what it depicts.

Also with regard to forms, some important semantic information is missing. While input elements do have a "label" that is visible to the user (e.g., "Select Your Origin" for the radio button controls in Travel Booking, or "Number of Children" in front of the respective input field in Booking Kärnten), this label cannot be identified programmatically (i.e., by assistive technology) since it is not marked as a label in the source code (e.g., by using the `<label>` tag). This becomes an issue if, for example, a blind user navigates through the form using the "tab" control to move the focus from one input to the next. If the label of the focused input can be programmatically determined, the label is read by the screen reader as soon as the input receives focus. Otherwise, only the input type (e.g., text field, radio button) is indicated to the user, but the user does not know what kind of information is expected (e.g., a name, a date, the number of children). The user would have to instruct the screen reader to read the content surrounding the input field to determine what its purpose is.

Another very similar issue with forms in the generated applications is that the `name` attribute of input fields is not descriptive. While screen readers could fall back to the `name` attribute in case there is no `label`, this would not help if the `name` is not descriptive.

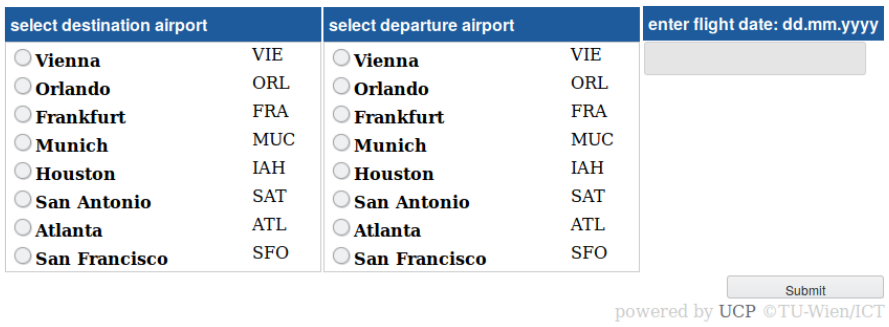
A further issue is that radio button groups in the prototypes are not surrounded by a `<fieldset>` tag. Screen readers indicate grouped radio buttons based on this tag and provide the user with a description that matches the `legend` belonging to the `fieldset`. If this information is missing, the radio buttons within a group are not perceived as belonging together and the user might miss some options.

Using HTML and CSS validators, some parsing issues were found, e.g., that IDs are assigned multiple times to different elements, or that attribute values are not surrounded by quotation marks.

Usage of Certain Interface Components

Two of the interface components being used in UCP currently are not accessible: the date picker and the tab control.

The date picker cannot be operated by a keyboard, since the date picker's input field is read-only per default and the trigger for opening the picker menu is an `onClick` handler, which cannot be activated using a keyboard. Figure 3.13 shows how the date picker is displayed. It appears to be disabled, hence it is not clear to the user that he or she can actually pick a date. Additionally, since the input field does not have a label and is marked as a text input field (`<input type="text" .../>`), a blind user does not know which input is expected. When the date picker receives focus, there is no visual indication that it is focused. To sum up, the date picker can neither be used by people with disabilities requiring them to rely on keyboard-only input, nor by people who need screen readers to read the contents to them.



select destination airport		select departure airport		enter flight date: dd.mm.yyyy
◯ **Vienna**	VIE	◯ **Vienna**	VIE	
◯ **Orlando**	ORL	◯ **Orlando**	ORL	
◯ **Frankfurt**	FRA	◯ **Frankfurt**	FRA	
◯ **Munich**	MUC	◯ **Munich**	MUC	
◯ **Houston**	IAH	◯ **Houston**	IAH	
◯ **San Antonio**	SAT	◯ **San Antonio**	SAT	
◯ **Atlanta**	ATL	◯ **Atlanta**	ATL	
◯ **San Francisco**	SFO	◯ **San Francisco**	SFO	

Submit

powered by UCP ©TU-Wien/ICT

Figure 3.13: The Flight Booking application viewed in Firefox. The date input field appears to be disabled, but a date picker actually opens when clicking on it.

The tab control component misses semantic information that indicates for the tabs, which tab is currently active, and whether a tab content is currently visible or hidden. Such semantic information makes it possible for assistive technology and user agents to interpret the controls correctly and can be achieved by applying the corresponding aria roles and attributes (e.g., `role="tablist"`, `role="tab"`, `role="tabpanel"`, `aria-controls`, `aria-selected`, `aria-hidden`). Blind users, for example, currently do not know which tab is active, since the only indication for a tab being active is its background color (which is white for inactive tabs, and gray for the active tab).

The handling of user input errors could be improved. Some required form fields are not marked as such (e.g., the radio buttons). If the user does not fill in a required field or there is any other input error, and the user clicks on submit, an alert dialog appears that indicates an error. The alert dialog only displays one error at a time, i.e., if the user made multiple mistakes, he or she has to submit the form multiple times and will see an alert dialog after each submit describing the next error. This is cumbersome, since it already takes many disabled people quite some time to fill out the form, read and understand the content, and so on and having to correct one error at a time and resubmitting prolongs the process further. The behavior after dismissing the alert dialog is not consistent, too. In some cases, the focus is automatically set to the erroneous input field mentioned in the alert dialog, in other cases it is not. Also, if the user dismissed the alert dialog without reading its content, there is no other indication of the error and the user would only see its description again after resubmitting the form. It would be better to provide a textual description of all errors with links to the corresponding input fields, which are themselves marked visually and semantically as being erroneous.

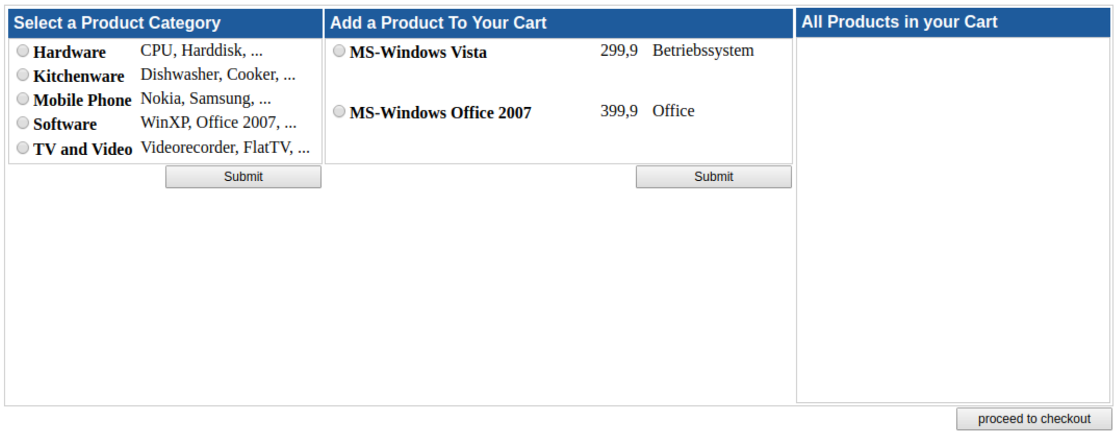
Keyboard-only Operability

Besides the date picker, which is not operable using just a keyboard, an `onClick` handler is attached to the submit buttons that cannot activated when operated with a keyboard. Interestingly, when using a mobile device, the return control of the on-screen keyboard triggers a submit of the form. While this means that a form submit can be triggered on mobile devices (i.e., without a mouse and only using the on-screen keyboard), this also means that the form is submitted unexpectedly when pressing the return key. Typically, the user would expect that as long as there are further input fields, the next input field gains focus and the user can continue filling in the form.

Issues Faced When Using a Screenreader

Blind users that depend on screen readers for accessing the applications face several issues. One of the reasons is that an old layout technique is used to display the content: HTML layout tables. This technique uses `<table>` elements to create page layouts by dividing content into rows and columns of the table. While this is per se not a violation of the WCAG guidelines, some screen readers cannot handle layout tables well. Screen readers try to linearize such layouts in order to determine the reading order of content. The result of this linearization might not be the same as the visual reading order, which becomes a problem when the reading sequence is not meaningful anymore. Moreover, some screen readers (like Google Talkback) read out loud at which row or column of the table the user currently is at (e.g. “row 1, column 1, checkbox not activated”), which makes it really hard for the user to recognize the actual content of the Web page. An example of a screen reader’s reading sequence being not meaningful occurs in the Travel Booking application: On the second screen, the submit button is read first, followed by “Select a Hotel” and the cancel button. Sometimes, after clicking on the submit button, the screen reader does not automatically read any content on the next screen.

The Shopping application contains an additional problem for screen reader users. After each form submit, the page reloads, and a new column of content appears at the right of the screen in addition to the existing content (after choosing the product category, the products to choose from appear right next to them, and so on). Since there is a page reload, the screen reader first reads all the previously existing content before reading the newly added information. In addition, all previously selected radio buttons are deselected. If a blind user, for example, has chosen a product category, then the page reloads and the screen reader would read the available product categories again and would indicate that none of the products is chosen. Up to this point, the user would think that something went wrong and no product category was selected. Only after the screen reader continued reading the available products, the user would know that there was no error. Figure 3.14 illustrates this issue. In this example, the user has already selected the “Software” product category. After pressing the “Submit” button below the product categories, the page reloads and shows the products in the second column. Now, the radio button for the “Software” product category is deselected and the screen reader starts reading from the beginning of the Web page due to the page reload.


Select a Product Category

○ **Hardware**	CPU, Harddisk, ...
○ **Kitchenware**	Dishwasher, Cooker, ...
○ **Mobile Phone**	Nokia, Samsung, ...
○ **Software**	WinXP, Office 2007, ...
○ **TV and Video**	Videorecorder, FlatTV, ...

Submit

Add a Product To Your Cart

○ **MS-Windows Vista**	299,9	Betriebssystem
○ **MS-Windows Office 2007**	399,9	Office

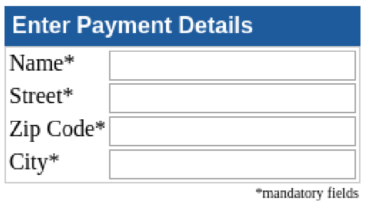
Submit

All Products in your Cart

proceed to checkout

Figure 3.14: The Shopping application after selecting the “Software” product category.

Mandatory fields (which occur in the Travel Booking application, see Figure 3.15), are only marked using an asterisk (*). They do not have any semantic markup that identifies them as required fields (i.e., the `required` or `aria-required="true"` attributes). The explanatory text at the bottom of the input fields (“*mandatory fields”) is not directly next to one of the labels. By the time the screen reader reaches this text (and it would not even reach it if the user just moves the focus from one input to the next), the user might not relate this information to the labels that were read before, and hence, the user would not understand that the corresponding input fields are mandatory. For users with impaired vision, the explanatory text might be too small.


Enter Payment Details

Name*	
Street*	
Zip Code*	
City*	

*mandatory fields

Figure 3.15: Indication of mandatory fields in the Travel Booking application

3.3.2 Accessibility Issues Failing Level AA

Level AA is required by the EN 301 549 standard. The discovered issues concerning this conformance level mainly address the responsiveness of the layout to various screen sizes and content magnification. When using the zoom function of the browser, horizontal scroll bars appear, since the layout is a non-responsive table layout. The WCAG requires content to be zoomed to an equivalent width of 320 CSS pixels without requiring to scroll on more than one axis, since the impact of two scrolling axis would increase reading effort 40 to 100 times [CCGV08]. In the Shopping application, the layout requires the user to scroll horizontally after choosing a product category because the additional content (for choosing a product) does not appear at the bottom but on the right of the screen (see Figure 3.16). Even more problematic is that the user might not even know that additional content has appeared until he or she has scrolled horizontally, because the additional content is not visible at first.

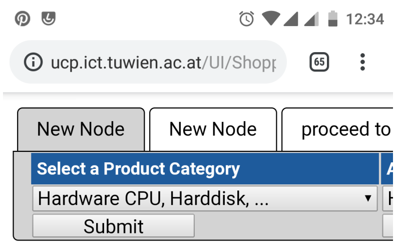
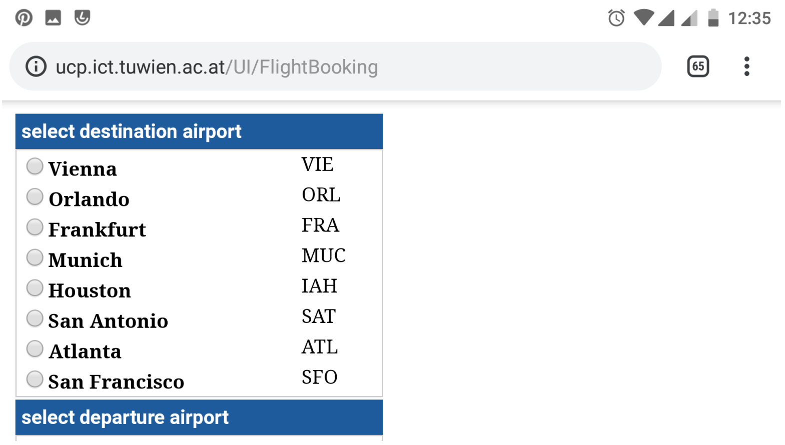

12:34
ucp.ict.tuwien.ac.at/UI/Shopp
New Node
New Node
proceed to
Select a Product Category
Hardware CPU, Harddisk, ...
Submit

Figure 3.16: The Shopping application viewed on a smartphone: to view available products, the user has to scroll horizontally

When switching from the portrait mode to the landscape mode in Flight Booking, the content does not adapt to the new screen size (see Figure 3.17). There is a lot of white space on the right of the screen that could be used to enlarge the content in order to fit information on the screen or to make spaces between elements bigger in order to make sizes of pointer targets bigger.


12:35

ucp.ict.tuwien.ac.at/UI/FlightBooking

select destination airport

◯ **Vienna**	VIE
◯ **Orlando**	ORL
◯ **Frankfurt**	FRA
◯ **Munich**	MUC
◯ **Houston**	IAH
◯ **San Antonio**	SAT
◯ **Atlanta**	ATL
◯ **San Francisco**	SFO

select departure airport

Figure 3.17: The Flight Booking application in landscape mode on a smartphone

Regarding the tab control, the naming and positioning of the tabs is not always coherent. In the Shopping application, the tabs are named “C Q_ Offer Product Categories”, “New Node”, and “proceed to checkout” at the beginning. After choosing a product category, the first tab is renamed to “New Node”. These names do not describe the contents of the tab and are very confusing to the user.

3.3.3 Accessibility Issues Failing Level AAA

Level AAA is the highest conformance level of the WCAG, and it is not required by any European regulation. In fact, the W3C recommends to not require level AAA by law since it is not achievable for some types of content [CGV+18]. Guidelines on this level are very likely to impact the “look and feel” of an application.

In the generated applications, five success criteria of level AAA were not met. While the contrast requirements of level AA were met (4.5:1 for text and images of text, 3:1 for adjacent colors of user interface components), the required contrast for level AAA, which is at least 7:1 for text and images of text, is not achieved for headlines and date pickers. The intention of this guideline is that even people with low vision that do not use contrast-enhancing assistive technology can read the content.

Headlines do not have semantic annotations (i.e., `<h1-h6>` tags). The only way to identify them as headlines is by use of color (white text on dark blue background). This means that screen readers cannot provide quick ways of navigating a Web page by moving from one headline to the next, the user has to listen to the whole content to know what the page is about.

The target size for clickable (focusable) elements is too small for level AAA, which requires them to be at least 44x44 pixels. This is especially relevant for UI components that are used frequently, are hard to reach (e.g., they are placed near the screen edges), are part of a sequential task or trigger functionality that cannot easily be undone [CGV+18].

There is no context-sensitive help available. It would be helpful if the required format is indicated for, e.g., date or credit card input fields (by an example given next to the input field).

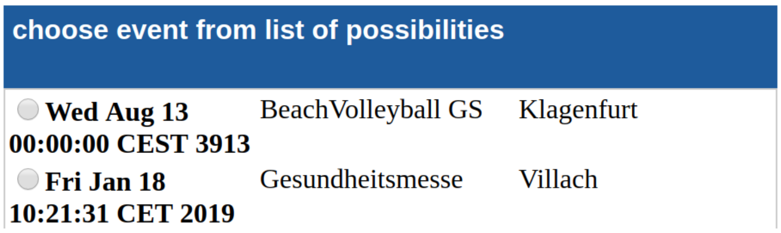
There is no possibility to navigate between the generated applications’ screens. Especially people with a short attention span benefit from an indicator of their current location (e.g., a breadcrumb or a site map), so that they know how many steps there are in the process and at which step they currently are. A navigation between the screens would also be helpful with regard to error prevention, since users could go back and forth to review and correct their inputs.

3.3.4 Other Discovered Issues That Impact Accessibility

During evaluation, some potential issues were discovered that do not necessarily violate WCAG success criteria, but still can be problematic. They are not necessarily violations, because there is some room for interpretation of the guidelines that sometimes make it hard to decide whether something is a violation of the guidelines or not. For example, page titles are not very descriptive, since each page’s title (i.e., the `<title>` tag’s content) simply is the name of the generated application. Descriptive page titles should indicate the current location within an application, so that the user does not have to skim through the content to know what the page is about. Even though the success criterion “2.4.2 Page Titled” is not violated, it is recommended by the W3C to provide more detailed titles that describe the contents of each page [CCGV08].

Two warnings were given by the CSS validator: (1) In case a foreground color (e.g., text color) is defined, a background color should also be defined and vice versa, and (2) a generic font-family (e.g., serif or sans-serif) should be defined as a fallback option if the preferred font could not be applied (e.g., if the font is not installed on the user’s system). Missing foreground or background colors can pose a problem if the content is, for example, resized and suddenly there are overlaps of certain regions that do not have enough color contrast. If there is no generic font-family defined, unexpected layout issues could occur, e.g., due to different character spacings.

In the Booking Kärnten application, date formats are inconsistent and sometimes not easily readable, since they are too detailed (like in the example given in Figure 3.18). The inconsistency can confuse users and people who are not familiar with the date format depicted in the figure might not understand it at all.


choose event from list of possibilities

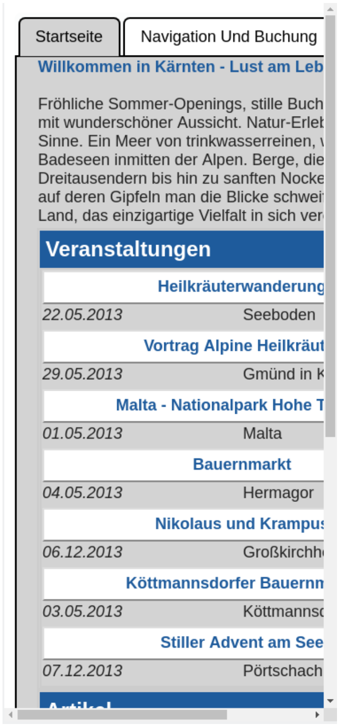
○ **Wed Aug 13 00:00:00 CEST 3913**	BeachVolleyball GS	Klagenfurt
○ **Fri Jan 18 10:21:31 CET 2019**	Gesundheitsmesse	Villach

Figure 3.18: Unneccessarily detailed date format which is not easily readable (found in Booking Kärnten event search results)

3.4 Lessons Learned

The case study of UCP’s status regarding accessibility as analyzed revealed some interesting findings:

Generating the GUI at design-time without leaving some flexibility for the GUI to adapt itself to various context conditions is, in general, not sufficient with regard to accessibility. Take, for example, the browsers zoom functionality: when the user zooms the application, the GUI components are enlarged while keeping the current browser window size. When the GUI generation framework built the GUI based on a device specification with a certain screen size and uses fixed container and component sizes (as it is the case with UCP), at some zoom level, the user will have to scroll, likely even in two directions. Take, for example, the Accommodation Booking application viewed on a smartphone with a resolution of 375x810 pixels. It was tailored to devices with a screen width of 380 pixels. At 150% zoom level, the user already has to scroll both vertically and horizontally, as can be seen in Figure 3.19. This violates success criterion “1.4.10 Reflow” of WCAG 2.1, which requires that the user can view the application at a 400% zoom level without having to scroll in two directions. Using the current approach of UCP to generate GUIs that adhere to this rule would require to generate a tailored GUI for each of the combinations of defined screen sizes and zoom level, which obviously is infeasible. Moreover, the GUIs would need to be exchanged as soon as the browser’s zoom level is changed. Hence, a more flexible approach that dynamically adapts the GUI at run-time is needed.



Willkommen in Kärnten - Lust am Leb

Fröhliche Sommer-Openings, stille Buch
mit wunderschöner Aussicht. Natur-Erleb
Sinne. Ein Meer von trinkwasserreinen, v
Badeseen inmitten der Alpen. Berge, die
Dreitausendern bis hin zu sanften Nocke
auf deren Gipfeln man die Blicke schwei
Land, das einzigartige Vielfalt in sich ver

Veranstaltungen

	Heilkräuterwanderung
22.05.2013	Seeboden
	Vortrag Alpine Heilkräut
29.05.2013	Gmünd in K
	Malta - Nationalpark Hohe T
01.05.2013	Malta
	Bauernmarkt
04.05.2013	Hermagor
	Nikolaus und Krampus
06.12.2013	Großkirchh
	Köttmannsdorfer Bauernn
03.05.2013	Köttmannsc
	Stiller Advent am See
07.12.2013	Pörtschach

Figure 3.19: The Accommodation Booking application viewed at 150% zoom level on a smartphone with a resolution of 375x810 pixels

Another important issue to consider when developing a GUI generation framework is what kind of information is needed to be given by the models in order to make the GUI accessible. For example, elements that need to be in close proximity because they are tightly related need to be grouped together in the model (e.g., elements belonging to a field set within a form). This is especially relevant when parts of the GUI are reorganized dynamically as the screen size changes. There must be a way to identify which elements need to stay together.

Additionally, some components need certain kinds of semantic information to be accessible (e.g., alternative texts for images, indication of the language, which form fields are required and which are not). Such information must be incorporated into the model or otherwise be provided by the model developer.

If the information cannot be woven into the models, there must be a way to adapt the resulting GUI at some level (e.g., at CUI level), i.e., a semi-automatic approach of GUI generation. For example, consider the various screens of the Flight Booking application. The process of booking a flight is separated into multiple screens: on the first screen, the user specifies the departure and destination airport as well as the flight date. On the second screen, the desired flight is selected among several options. In the third screen, the user can choose a seat. Credit card and passenger details are supplied in the fourth screen and the fifth screen then presents the purchased ticket's number. Each screen should have a title or headline that outlines its contents. Which contents it contains can only be known at the time the GUI components are split among the screens. The way the components are distributed among the screens can differ between device specifications (e.g., two components which are shown on the same screen on a desktop computer can be split between two screens on a mobile phone).

Another aspect to be considered is which GUI libraries are being used. For example, the date picker component in UCP was taken from a library. Unfortunately, it has several shortcomings with regard to accessibility (see Section 3.3.1). The accessibility of GUI components from libraries has to be evaluated beforehand. Moreover, these components often expect a certain data format given to them, which has to be considered.

The GUI source code also needs to be generated in a way that allows users to adapt certain aspects of the GUI for their needs. For example, in a Web application, the font sizes should be given in relative units (typically `em` or `rem`), so that the user can increase them via his or her browser settings. Using the zoom function of the browser, a user should not be required to scroll in two directions (horizontally and vertically), but instead the content should adapt and restructure itself to fit the new zoom level.

In principle, two different kinds of modalities of the application contents need to be considered: the visual and the non-visual (i.e., auditive or haptic) representation. A sighted user can identify related information based on visual cues like, for example, proximity, color or shape. A blind user cannot make use of these cues but instead has to rely on semantic information about the content which can be interpreted by assistive technology (e.g., headlines, labels) and, especially, the order in which the content is presented.

It can be reasonable to restructure or replace certain GUI components based on context characteristics. UCP is already capable of organizing certain GUI elements into a tabbed view on small screens. This can save space on the screen, so that the user does not have to scroll that much. Moreover, the cognitive load tends to decrease, since there is less information shown at once.

CHAPTER 4

Improving the Accessibility of UCP

Based on the findings from the case study, several improvements for supporting the development of accessible applications via GUI generation using UCP were identified and built into the framework. The remainder of this chapter is organized as follows: Section 4.1 explains how the accessibility of the generated GUIs can be improved. Section 4.2 presents a proof-of-concept application that demonstrates the improvements, which were evaluated during a user study that is described in Section 4.3. Finally, Section 4.4 describes technologies being used and the implementation for these accessibility enhancements within UCP.

4.1 Accessibility Improvements

The major accessibility improvements that were identified can be grouped roughly into five categories: responsiveness, enriching semantic information, error handling, enabling customization by the end user, and widget replacement.

Besides these five types of improvements, which are especially relevant for GUI generation, several other accessibility enhancing measures can be implemented to address issues that are not directly relevant to GUI generation, but were introduced into UCP because accessibility originally was not a major concern. For example, all `onClick` handlers has to be removed in order to make the application accessible by keyboard only. Also the date picker component has to be replaced by one that can be operated via keyboard. The new date picker also must clearly indicate when it is focused and must not be a read-only input field as it currently is in UCP. The color contrast has to be improved for some components, and spaces between elements had to be enlarged when viewed on mobile devices.

4.1.1 Responsive Design

The table layout of UCP was replaced by RWD using `<div>` tags. This ensures that the layout adapts itself to the available screen width. The table layout in UCP uses fixed widths and heights for the table cells, which coercively leads to horizontal scrolling when zooming. Using RWD, the content will dynamically reflow when a user zooms into the application. For example, if two widgets which are shown side-by side do not fit next to each other anymore when zooming, they can be regrouped below each other. Additionally, font sizes and spacings can also be enlarged as the user zooms in, making it easier for the user to read text. Responsive design has also been found to increase accessibility in various other ways (as discussed in Chapter 2).

4.1.2 Enriched Semantic Information

UCP would benefit greatly from more detailed semantic information. When generating the GUI, this kind of information has to be derived from the models. For example, the generated application should indicate which language is used throughout its contents. Alternative texts for images should not just resemble their file names, but should actually describe what is depicted.

Headlines were be marked up accordingly. Studies show that header elements (using the `<h1-6>` tags) substantially improve accessibility for blind users. Watanabe [Wat09] found that blind users' task completion times were considerably reduced when navigating through Web pages with marked up heading elements as compared to ones without them. The seventh Screen Reader User Survey conducted by WebAIM [WAI17] found that 67.5% of their 1792 participants used headings as their primary resource for finding information on a Web page.

Another main area where UCP would profit from enriched semantic information is in forms. In UCP as given in the case study, mandatory fields are marked by an asterisk, but are missing semantic markup that indicates that. This semantic markup has to be added to the input fields to enable assistive technologies to recognize them as being required. All input fields need to have a semantic label (using the `<label>` tag). This not only helps assistive technology to interpret the purpose of the belonging input field, but also helps sighted users, because they can also click on the label to focus the input field. In UCP as given in the case study, the user has to click on the radio button itself to activate it. Since radio buttons are quite small, users with motor disabilities may find it hard to click on them. Labels make the target area much bigger and this makes it a lot easier for users to focus the input fields.

Finally, input fields that are closely related should be grouped into fieldsets. This is especially relevant for radio buttons and checkboxes, since assistive technology can provide users with additional information (i.e., indicating the number of options in a radio button fieldset). Furthermore, in some cases, it would be helpful to provide users with context-sensitive help when dealing with forms. For example, date formats vary greatly, not only depending on language and region, but also in verbosity (e.g., sometimes only the date is required, sometimes the time should also be provided, and sometimes, as for the credit card validity period, it is sufficient to only provide the month and year). A hint given to the user about the required format is beneficial for avoiding errors and hence steps required to correct them. These hints should be derivable from the models used to generate the GUI.

4.1.3 Error Handling

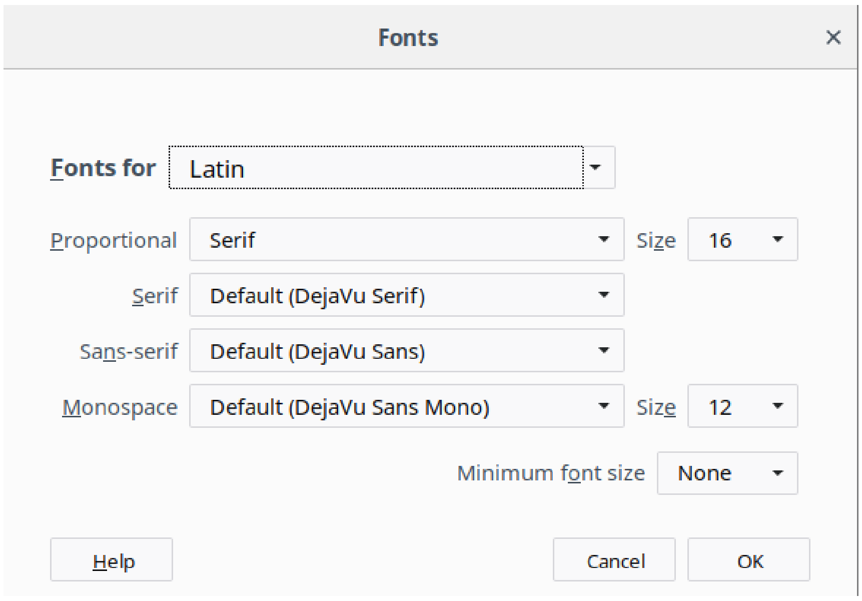
Form input errors could be handled better. Currently, only one error is shown in an alert dialog at a time. It would be better if all errors on a screen are indicated at once, so that the user does not have to press the "Submit" button each time just to find out that there are more errors. When activating the submit button, a list of error messages should be displayed. Each error message should describe which input field is concerned and why the input is erroneous. Each error message should also link to the according input field, so that by clicking on the error message the input field automatically gets focused. Next to the input field, there should be a further indication that it is erroneous.

4.1.4 Enabling Customization by the User

Sometimes users employ additional tools and techniques to make a Web interface more accessible to them and further customize it to their needs. Besides assistive technologies, they often use the browser's zoom capabilities to enlarge Web contents. Browsers typically support the enlargement of font sizes only. Users may also adapt their browser settings or use custom stylesheets to override, for example, default font styles and sizes, letter spacings, line widths and heights, or foreground and background colors. For example, as can be seen in Figures 4.1, in Firefox it is possible to change default style and size, the minimum font size shown on a Web page, text and background colors as well as unvisited and visited link colors and whether links should always be underlined. Additionally, there are various browser extensions that modify Web content to fit the user's individual needs (e.g., Google provides a Chrome plugin for enabling a high contrast color scheme on any Website[1]).

To make a Web GUI customizable for the user in this way, several measures have to be taken. To ensure that zooming Web content does not introduce scrollbars in two directions, the content needs to reflow to fit onto the screen without introducing horizontal scrollbars. This can be achieved by using responsive design. In order to be able to enlarge or minimize font sizes only, they have to be defined in relative units (like `em`, `rem`, or `%`). Using the expression `!important` in CSS should be avoided, since it elevates the specificity of the rule it was applied to and, therefore, could potentially override a matching rule defined in the user's custom stylesheet (if the user's custom rule is not marked as `!important` and the browser does not treat custom stylesheets as a priority).

[1]https://chrome.google.com/webstore/detail/high-contrast/djcfdncoelnlbldjfhinnjlhdjlikmph (Last Access: 2018-10-07)



Fonts

Fonts for: Latin

Proportional: Serif — Size: 16

Serif: Default (DejaVu Serif)

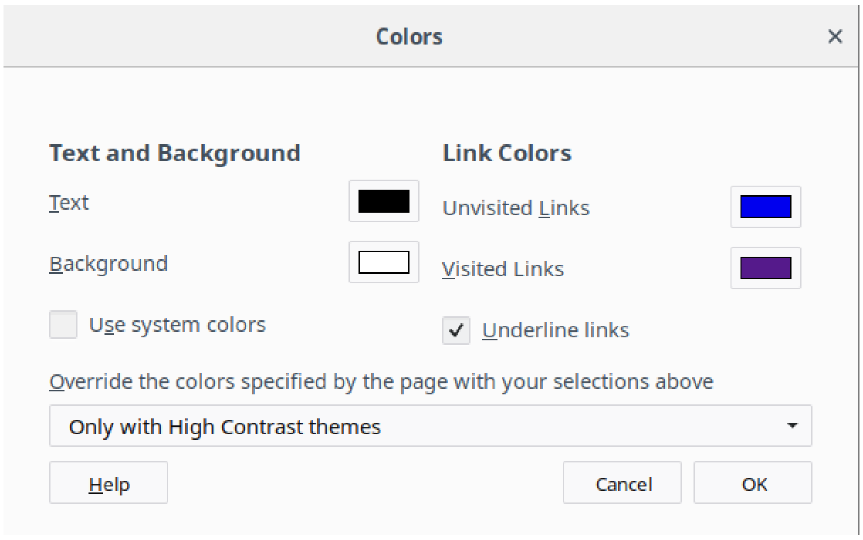
Sans-serif: Default (DejaVu Sans)

Monospace: Default (DejaVu Sans Mono) — Size: 12

Minimum font size: None

Help — Cancel — OK

(a) font settings



Colors

Text and Background

Text: ⬛

Background: ⬜

☐ Use system colors

Link Colors

Unvisited Links: 🟦

Visited Links: 🟪

☑ Underline links

Override the colors specified by the page with your selections above: Only with High Contrast themes

Help — Cancel — OK

(b) color settings

Figure 4.1: Browser settings for customizing default font styles, sizes and colors in Firefox

Additionally to these techniques, the table component in UCP may benefit from customizability. On small screens it may be helpful to provide the user with a possibility to select the columns that he or she wants to be displayed. This would reduce the amount of information that is shown at once and thus reduce cognitive load and the amount of scrolling needed. However, this possibility to select columns might also distract the user from the primary task (i.e., reading the data) and hence should only be shown if there are many columns.

4.1.5 Widget Replacement

Some GUI components work well for accessibility in certain situations, but not in others. For example, in the Accommodation Booking application, when viewed on a desktop computer, UCP shows a quite complex interface with a header image, a navigation bar on the left, and a main content box including two columns showing upcoming events and some recent articles (see Figure 3.9). For small devices, the same GUI would be hard to use, because the user would need to scroll in two directions. Thus, UCP already makes use of device tailoring and replaces the widgets with a tab view, showing at least two tabs (one or more for the current page and one for the navigation, see Figure 3.10).

Widget replacement can enhance accessibility by avoiding scrolling in two directions, saving space (so that the user does not have to scroll as much), hiding less used functionality, and showing a limited amount of information at once (and, therefore, reducing cognitive load). Besides the already implemented widget replacement (i.e., reorganizing page content into tabs), UCP could benefit from the following widget replacements.

Tabs to Accordion

When the tab view in UCP applications has too many tabs so that they cannot fit in one line, the user might be confused by that. In Figure 4.2, the active tab is not connected to its content anymore, and the user might miss this connection. In this case, it would be clearer to use an accordion widget, where the tabs are organized vertically instead of horizontally. Figure 4.3 shows an example of an accordion widget. Like in the existing tab view, in the accordion widget only one tab can be expanded at a time. The content is shown directly below its tab.

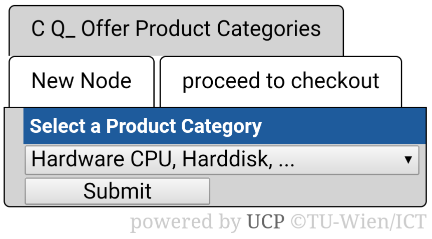

C Q_ Offer Product Categories
New Node
proceed to checkout
Select a Product Category
Hardware CPU, Harddisk, ...
Submit
powered by UCP ©TU-Wien/ICT

Figure 4.2: The Shopping application viewed on a smartphone in portrait mode, the tabs do not fit in one line

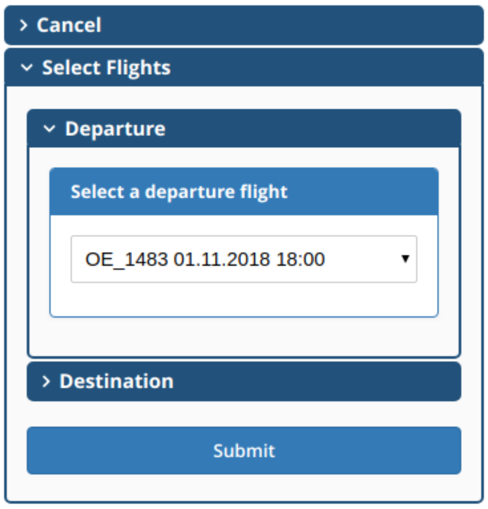


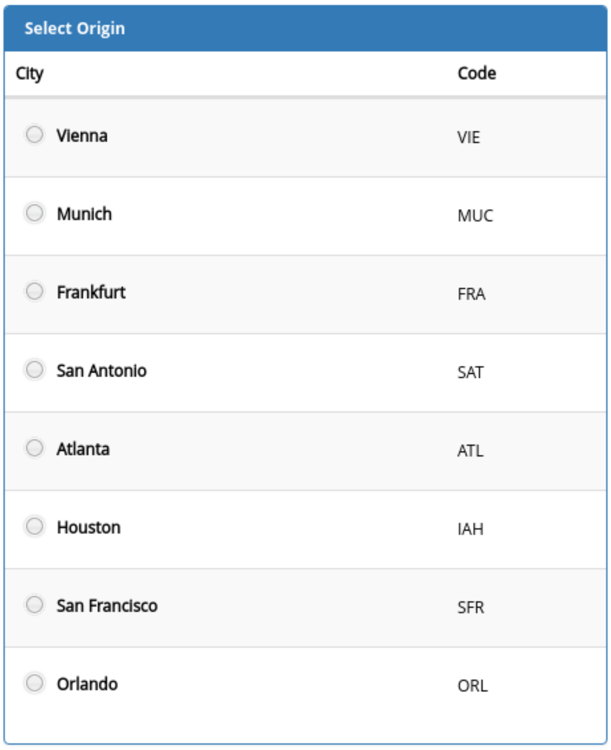
Figure 4.3: The accordion view aligns tabs vertically with the content shown directly below its tab

Radio Buttons to Dropdown

Radio button groups, especially when they contain many options, take up a lot of space. This can be an issue on smartphones, since the user would have to scroll a lot and does not see much else of the information other than these options. In such cases, it would be better to replace the radio buttons with another widget that does not take up that much space, like a dropdown.

Radio button groups and dropdowns each have their own advantages and disadvantages. Radio button groups are easier to operate for users with motor disabilities if their options have labels associated to them. As discussed in Section 4.1.2, labels enlarge the target size of an input field, since one can also activate the inputs by clicking on the labels. Radio button options are typically larger than dropdown options, and thus, dropdown options require more precise mouse movements (or touch interaction on mobile devices, respectively) to activate them. Additionally, there is one more action required (i.e., clicking or tapping on the dropdown) to view all available options. However, when there are a lot of options, a dropdown needs significantly less space, and thus, the size of a Web page is reduced. The user can see more of the screen and only expands the options as he or she needs. Figures 4.4 and 4.5 demonstrate this: In Figure 4.4, the options for selecting an origin in the Flight Booking application are displayed as radio buttons, while in Figure 4.5, the options are grouped into a dropdown.

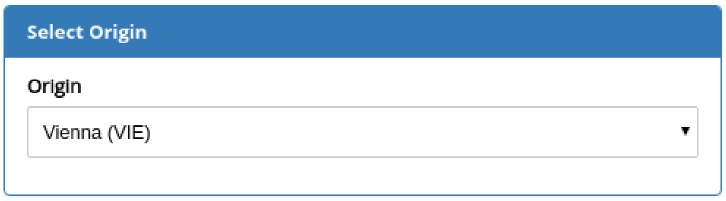
To use the best of both approaches, radio button groups should be replaced by dropdowns when there are more than six options. Six options seem to be a sensible choice because they can still fit on even a small screen. Thus, if there are at most six options, it makes sense to display them as radio buttons, since selecting an option does require less precise mouse movements than a dropdown. For more than six options, a dropdown can save



Select Origin

City	Code
○ Vienna	VIE
○ Munich	MUC
○ Frankfurt	FRA
○ San Antonio	SAT
○ Atlanta	ATL
○ Houston	IAH
○ San Francisco	SFR
○ Orlando	ORL

Figure 4.4: Selection of the origin airport, displayed as a radio button group



Select Origin

Origin

Vienna (VIE) ▾

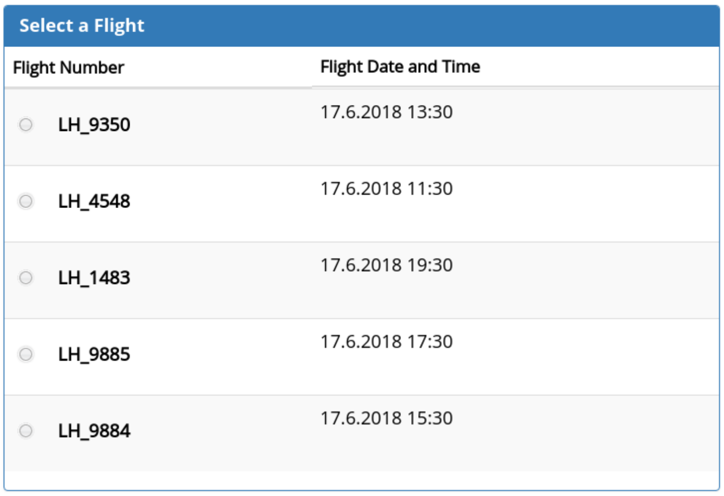
Figure 4.5: Selection of the origin airport, displayed as a dropdown

screen space. When displaying a dropdown, there should also be the possibility to search and auto-complete the options, by using a combined text input field and dropdown. This makes it easier to find the desired options among a large list of options, and users that are able to operate a keyboard can select an option quicker than by using a mouse.

Table to Stacked Table

Tables pose a special challenge on small screens. The more columns there are, the more the user has to scroll horizontally to view the data. If there are many rows, the user has to scroll vertically, too. To avoid scrolling in two directions, stacked tables can be used. In a stacked table, the columns are re-organized into labeled rows. To distinguish the original table rows, they have, for example, alternating background colors and are separated by thick horizontal lines.

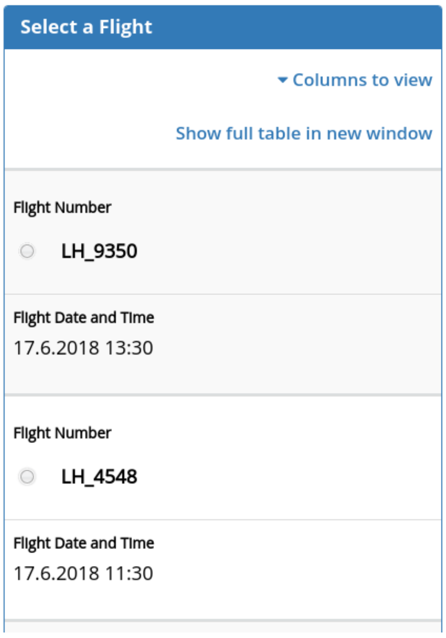
Figures 4.6 and 4.7 demonstrate this replacement. Figure 4.6 shows the options for selecting a flight displayed in a typical table view. If the columns cannot fit onto the screen without introducing horizontal scrolling, the table is replaced by a stacked table, as can be seen in Figure 4.7.



Select a Flight

Flight Number	Flight Date and Time
○ LH_9350	17.6.2018 13:30
○ LH_4548	17.6.2018 11:30
○ LH_1483	17.6.2018 19:30
○ LH_9885	17.6.2018 17:30
○ LH_9884	17.6.2018 15:30

Figure 4.6: Selection of a flight, displayed as a table on a desktop computer



Select a Flight

▾ Columns to view

Show full table in new window

Flight Number
○ LH_9350

Flight Date and Time
17.6.2018 13:30

Flight Number
○ LH_4548

Flight Date and Time
17.6.2018 11:30

Figure 4.7: Selection of a flight, displayed as a stacked table on a smartphone

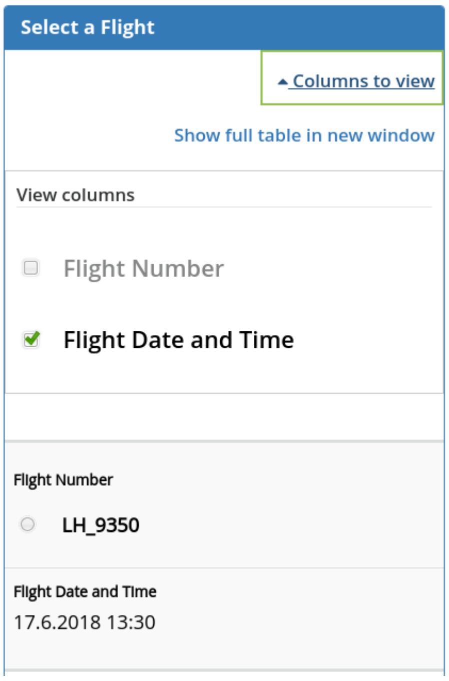
Now each column from the table is displayed as a row, in which the column title is displayed as a headline and the column content is shown below the headline. The table columns are now separated by a vertical line. Each table row is separated by a thicker line to distinguish it. Additionally, the alternating background colors help to identify the table rows.

In the example of Figure 4.7, it is also possible to show the full table in a new window (to provide the typical tabular view if the user prefers it over the stacked table view and does not mind if there is horizontal scrolling required).

Hiding Less Used Functionality or Information

The W3C recommends to hide less used functionality or information in order to reduce cognitive load and to find relevant content more easily[2].

In UCP, one place where this can be helpful is in tables (and stacked tables). Sometimes the user is not interested in all the information displayed in the table. Especially in stacked tables, less information can be helpful, since it leads to less scrolling and quicker skimming of the content. Figure 4.8 shows how this can be accomplished. The (stacked) table contains an option to select the columns that should be viewed. The user can select all the columns he or she wanted and deselect the others.


Select a Flight

▴ Columns to view

Show full table in new window

View columns

☐ Flight Number

✓ Flight Date and Time

Flight Number

○ LH_9350

Flight Date and Time

17.6.2018 13:30

Figure 4.8: Table columns may be shown and hidden as desired by the user, columns required for user interaction are always visible

[2]https://www.w3.org/WAI/people-use-web/tools-techniques/#presentation (Last Access: 2018-06-17)

However, this ability to select columns takes up some space of the screen and might distract the user from his or her main task (i.e., selecting a flight in this case), so this option might only be useful if the table has many columns.

4.2 Proof-of-Concept Application

Before actually implementing accessibility improvements within the UCP framework, a proof-of-concept application was developed, which demonstrates the improvements. In a user study, it was compared with the current approach of UCP. This was done in order to get feedback early on, since adapting the implementation based on the insights gained from the evaluation would have been more costly within the UCP framework.

There have been three versions of the proof-of-concept application. In the first version, a navigation was provided. Because it became clear early on that implementing such a navigation was technically not feasible in UCP without major enhancements of the runtime framework, the navigation was excluded in later versions of the proof-of-concept application. The second version of the proof-of-concept application (which included everything from the first version but the navigation) was then used for the user study. After the user study, two widget replacements, "table to stacked table" and "radio buttons to dropdown" have been added to the third version of the proof-of-concept application to incorporate lessons learned during the user study and further research.

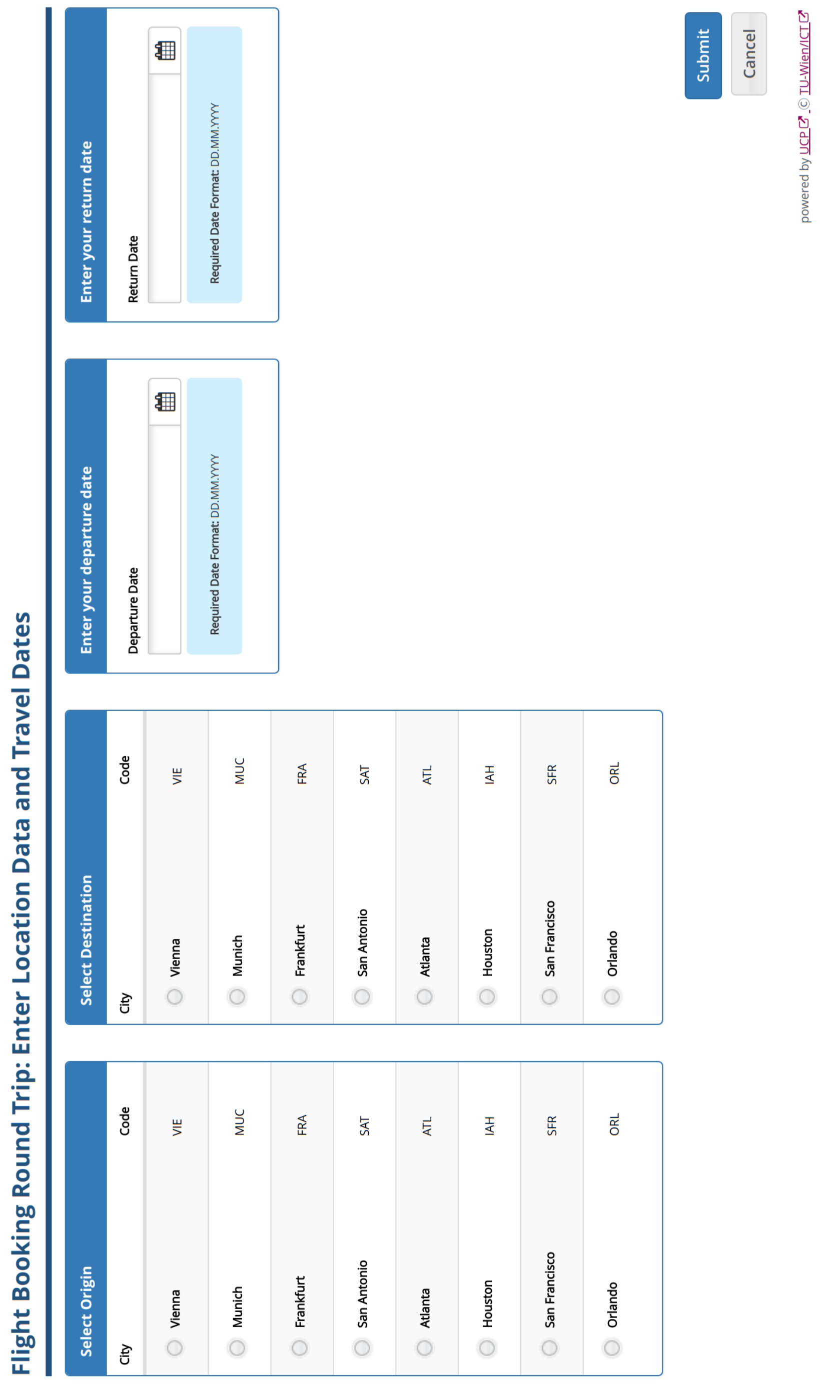
The proof-of-concept application is based on the Flight Booking Round Trip application. The decision to use this application was based on the fact that it incorporates most of the features that UCP is capable of while at the same time being free from customizations like custom widgets or custom CSS. Custom widgets or CSS files are outside the scope of this diploma thesis, since they are provided by a (human) widget designer and hence cannot be controlled by the framework. The Flight Booking Round Trip application does not include a tab view or images, although tab views have been addressed in the implementation of accessibility improvements in UCP.

The remainder of this section explains how the proof-of-concept application enhances accessibility in comparison to the given implementation of UCP.

4.2.1 Overall Structure of the Application and Responsiveness

Figure 4.9 shows the proof-of-concept application viewed on a desktop computer.

The application is fully responsive, i.e., it reflows its contents when the screen gets smaller. Figures 4.10 and 4.11 demonstrate this behavior on a tablet screen and a smartphone screen, respectively. When the screen gets smaller, at first, the two boxes containing the date pickers for selecting the departure and return date break into a new line below the boxes containing the tables for selecting origin and destination. As the screen gets smaller again, the four boxes align into a one-column-layout. Additionally, the tables are replaced by a dropdown to save more screen space. Hence, horizontal scrolling is



Flight Booking Round Trip: Enter Location Data and Travel Dates

Select Origin

City	Code
○ Vienna	VIE
○ Munich	MUC
○ Frankfurt	FRA
○ San Antonio	SAT
○ Atlanta	ATL
○ Houston	IAH
○ San Francisco	SFR
○ Orlando	ORL

Select Destination

City	Code
○ Vienna	VIE
○ Munich	MUC
○ Frankfurt	FRA
○ San Antonio	SAT
○ Atlanta	ATL
○ Houston	IAH
○ San Francisco	SFR
○ Orlando	ORL

Enter your departure date

Departure Date

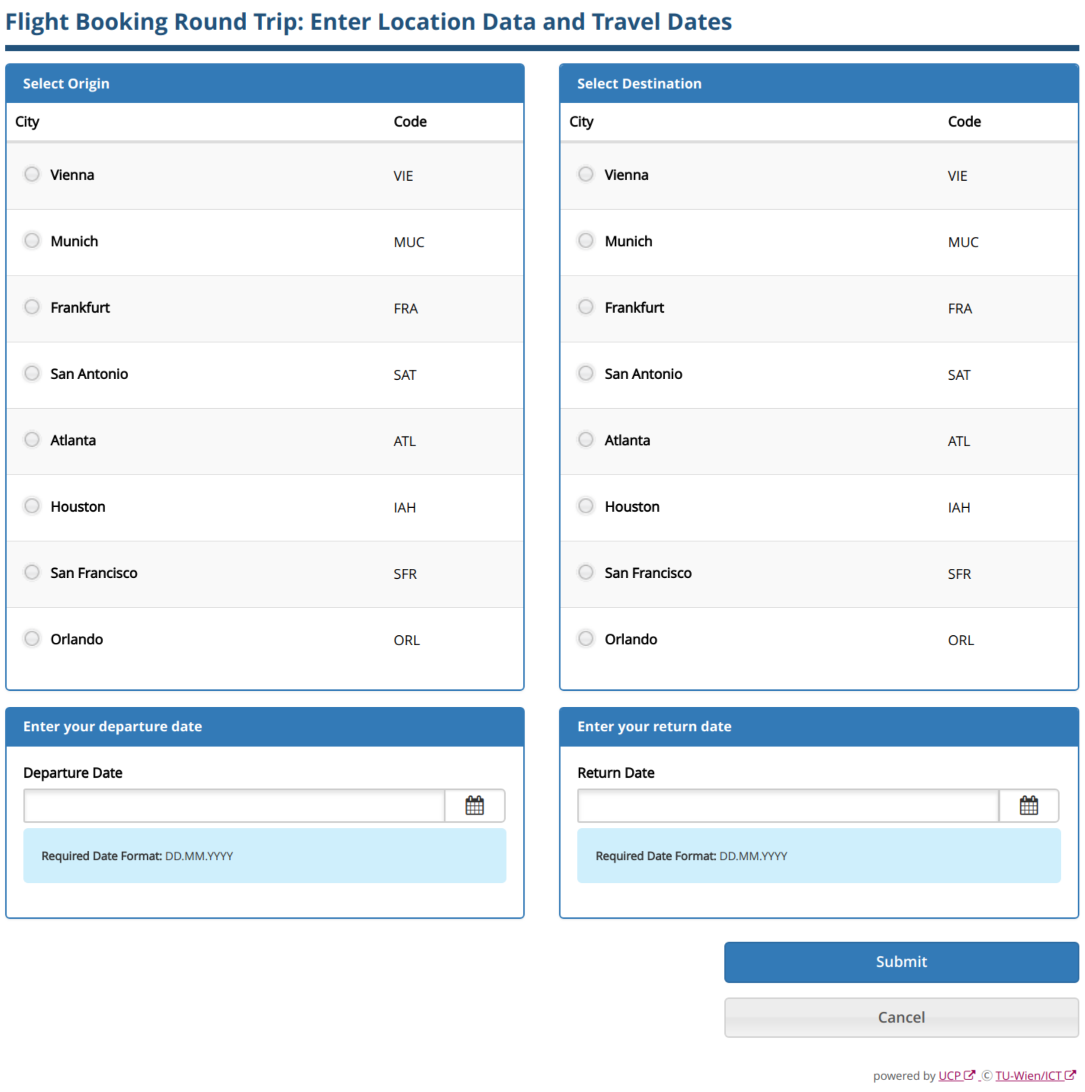
Required Date Format: DD.MM.YYYY

Enter your return date

Return Date

Required Date Format: DD.MM.YYYY

Figure 4.9: The first screen of the proof-of-concept application viewed on a desktop computer with a resolution of 1280x800px


Flight Booking Round Trip: Enter Location Data and Travel Dates

Select Origin

City	Code
Vienna	VIE
Munich	MUC
Frankfurt	FRA
San Antonio	SAT
Atlanta	ATL
Houston	IAH
San Francisco	SFR
Orlando	ORL

Select Destination

City	Code
Vienna	VIE
Munich	MUC
Frankfurt	FRA
San Antonio	SAT
Atlanta	ATL
Houston	IAH
San Francisco	SFR
Orlando	ORL

Enter your departure date

Departure Date

Required Date Format: DD.MM.YYYY

Enter your return date

Return Date

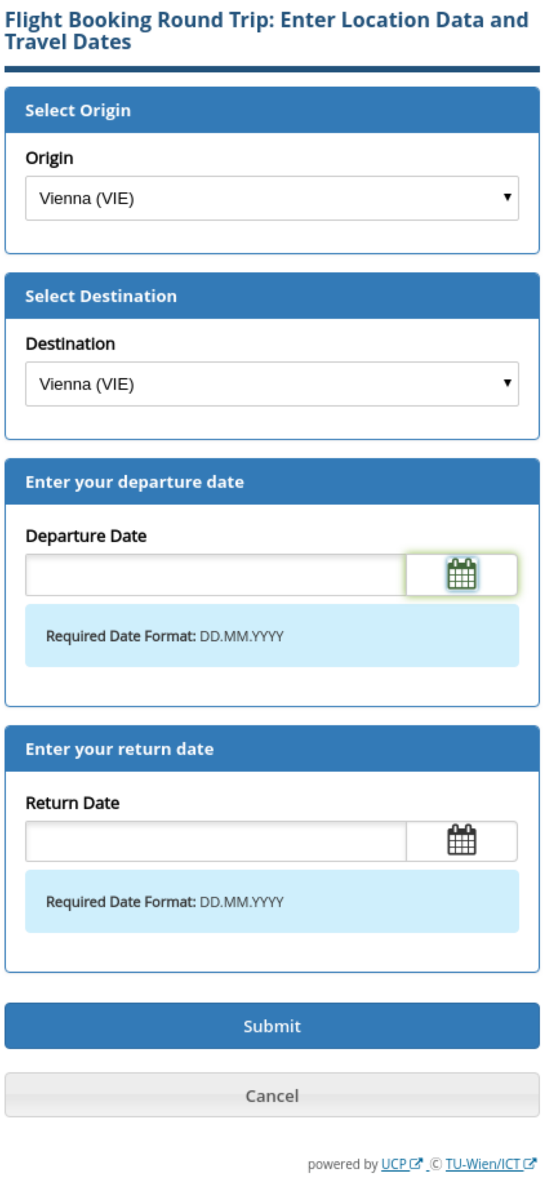
Required Date Format: DD.MM.YYYY

Figure 4.10: The first screen of the proof-of-concept application viewed on a tablet with a resolution of 1024x1366px

avoided (by reflowing the elements) and vertical scrolling is minimized (by replacing widgets where possible).

In comparison to the original Flight Booking Round Trip application (see Figure 4.12]), the spaces between elements are bigger. Spaces between the boxes separate them more clearly, and spaces between input elements (along with the labels) increase the click target.

A headline was added to the top (marked by the `<h1>` tag), which includes the name of the application and describes what the current screen is about. It is especially helpful for


Flight Booking Round Trip: Enter Location Data and Travel Dates

Select Origin

Origin

Vienna (VIE)

Select Destination

Destination

Vienna (VIE)

Enter your departure date

Departure Date

Required Date Format: DD.MM.YYYY

Enter your return date

Return Date

Required Date Format: DD.MM.YYYY

Submit

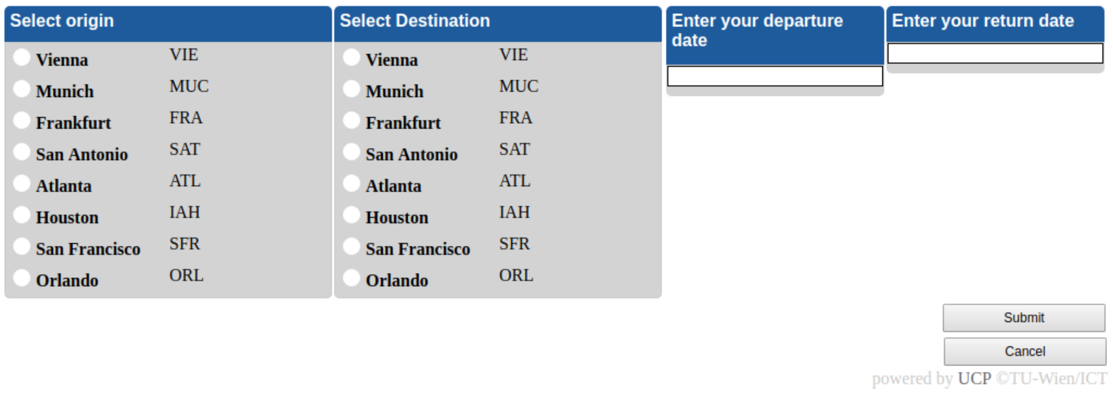
Cancel

powered by UCP © TU-Wien/ICT

Figure 4.11: The first screen of the proof-of-concept application viewed on a smartphone with a resolution of 375x812px

blind users, since screen readers will read it first when accessing the Web page and give the user an idea of what it is about. Further, the headlines of the boxes are marked up as such using the <h2> tag, so that it is possible for screen reader users to quickly skim through the headlines and figure out which logical sections there are on the Web page.

In case JavaScript is deactivated in the user's browser, a hint will be displayed that indicates that certain functionalities will not be available.



Select origin		Select Destination		Enter your departure date	Enter your return date
Vienna	VIE	Vienna	VIE		
Munich	MUC	Munich	MUC		
Frankfurt	FRA	Frankfurt	FRA		
San Antonio	SAT	San Antonio	SAT		
Atlanta	ATL	Atlanta	ATL		
Houston	IAH	Houston	IAH		
San Francisco	SFR	San Francisco	SFR		
Orlando	ORL	Orlando	ORL		

Figure 4.12: The first screen of the Flight Booking Round Trip application viewed on a desktop computer

HTML5 sectioning elements and ARIA landmarks are used to enhance the semantic information provided to screen readers. The application headline at the top (`h1`) is wrapped with a `<header>` element that has the `banner` landmark. These elements typically appear at the top of the page and contain content that identifies what the whole Web page is all about. This can include, e.g., page titles, logos, or search bars. At the bottom, there is a `<footer>` element with the `contentinfo` landmark that wraps the text "powered by UCP ©TU-Wien/ICT". This text is included in all applications generated by UCP and does not have a logical connection to the rest of the application, which is why it is contained within the footer. The rest of the application is wrapped within the `<main>` element.

In the footer, there are two external links, one leading to the UCP project Website, the other leading to the institute Website of ICT at TU Wien. These links are underlined (which is a widely-known convention for marking links as such), and are additionally appended by an icon to indicate that they are external links. When these links have already been visited before, they are colored in purple, otherwise their color is blue. Moreover, the color contrast has been enhanced for the whole text in the footer. Additionally, there is a hidden text that will only be read by screen readers which says "This link opens a new window or tab".

4.2.2 Visible Focus

In the proof-of-concept application, the focus is clearly visible for every element. In the Flight Booking Round Trip application, this was a problem especially for the date pickers, which did not indicate focus at all.

As demonstrated in Figure 4.13, when hovering an input element or its label with the mouse, a green shadowed border appears around the input element.

When clicking into the input element or focusing the input element using a keyboard, a

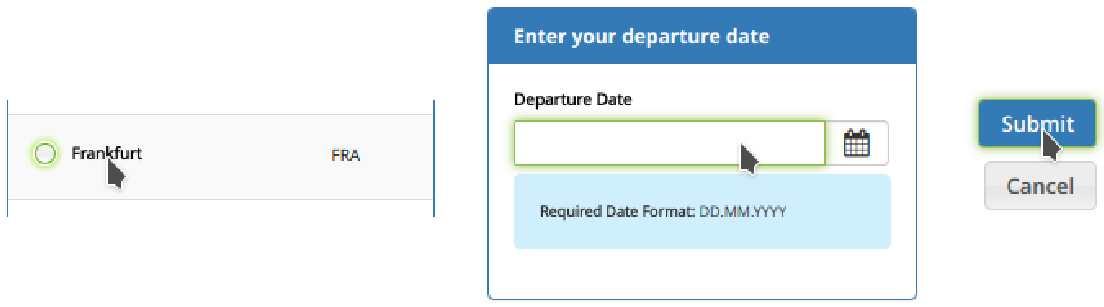


Figure 4.13: Three examples of the hover effect in the proof-of-concept application

two pixel thick green border appears around the input element, which can be seen in Figure 4.14. In case of radio buttons, the border also wraps around the radio button's label.

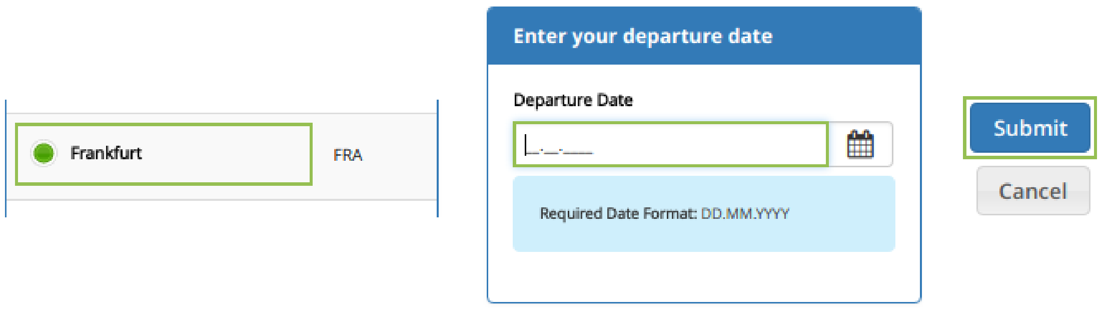


Figure 4.14: Three examples of the focus effect in the proof-of-concept application

4.2.3 Forms

Labels have been connected to their input fields using the `<label>` tag along with its `for` attribute to specify its corresponding input field. Where labels have been missing (e.g., for the date pickers on the first screen), they have been added. Connecting the labels to their input fields not only is required for screen readers to be able to read the purpose of an input field, there is also the advantage that the user can now also click on the label to focus its input field (as opposed to just click on the input field itself). This is especially helpful for radio buttons. In the original Flight Booking Round Trip application you need to click on the radio button itself, which is quite a small target. By being able to also click on its label to activate the radio button, the click target becomes much larger (see the first example in Figure 4.14, where everything inside the green rectangle can be clicked to activate the radio button for selecting "Frankfurt").

Radio button groups and groups of input fields have been surrounded by the `<fieldset>` tag to semantically group them. Each fieldset contains a `<legend>` tag that describes the input group (e.g., “Select origin” or “Enter your credit card details”).

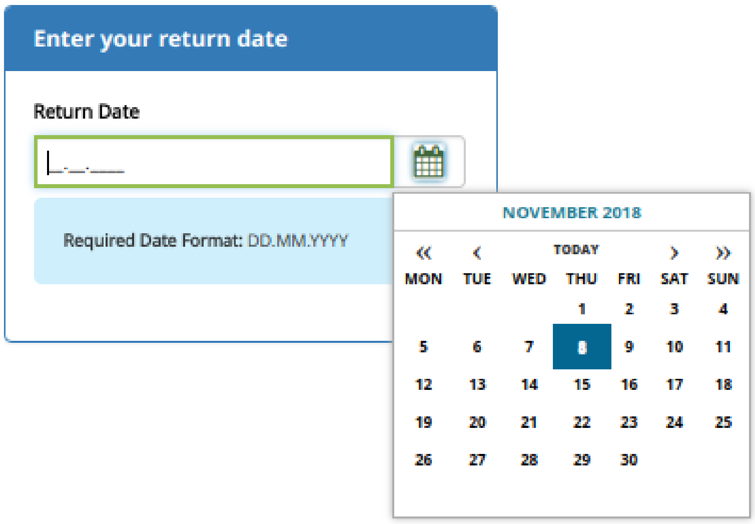
Input fields and the space between them are bigger in general and have more padding to make them clickable more easily. Radio buttons have more space between them and are replaced by dropdowns when there are more than six options and the screen width is smaller than 992 pixels.

The submit and cancel buttons are visually differentiated to better distinguish them. Users are less likely to accidentally activate the wrong button, since the submit button (the primary action) is colored in blue, while the cancel button has a light gray background.

Date Pickers

Input hints were added to the date input fields to describe the expected date format. They are connected to their respective input field by the ARIA attribute `aria-describedby`. Additionally, the jQuery plugin “Jasny Bootstrap Input Mask”[3] was used to add an input mask to the date picker input field. The input mask (i.e., “__.__.____”, which is also depicted in the second example shown in Figure 4.14) corresponds to the input hint and allows the user to put in just the numbers (i.e., there is no need to press the “.” key).

For the date picker itself, a new library has been chosen: Yet Another DatePicker[4]. Next to the date input field there is a button showing a calendar icon. By clicking on the button, a pop-up with the date picker opens (see Figure 4.15).


Enter your return date

Return Date

Required Date Format: DD.MM.YYYY

NOVEMBER 2018

MON	TUE	WED	THU	FRI	SAT	SUN
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

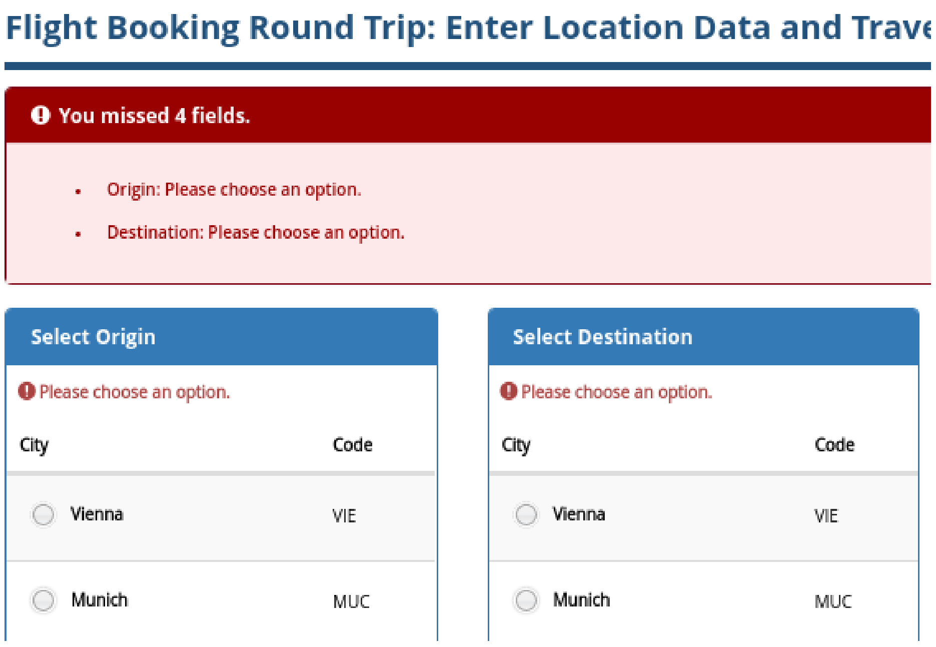
Figure 4.15: Date picker in the proof-of-concept application

[3]https://www.jasny.net/bootstrap/javascript/#inputmask (Last Access: 2018-11-03)
[4]https://freqdec.github.io/datePicker/ (Last Access: 2018-11-03)

The date picker can be easily operated by keyboard: there are shortcuts for choosing the previous or next day / week / month / year and a selection can be confirmed using the `Enter` key. Exiting the pop-up without a selection can be done using the `Escape` key, in which case the focus is given back to the date input field. As opposed to the date picker used in the original Flight Booking Round Trip, the currently hovered day is indicated visually by a dark blue background.

4.2.4 Error Handling

Figure 4.16 shows an example of error messages in the proof-of-concept application.



Flight Booking Round Trip: Enter Location Data and Trave

You missed 4 fields.

- Origin: Please choose an option.
- Destination: Please choose an option.

Select Origin

Please choose an option.

City	Code
Vienna	VIE
Munich	MUC

Select Destination

Please choose an option.

City	Code
Vienna	VIE
Munich	MUC

Figure 4.16: Indication of form errors in the proof-of-concept application after pressing the “Submit” button

In case there are form errors after pressing the submit button, a red box appears below the application’s headline. The red box is focused as soon as it appears, so that a screen reader will start reading it. Its title indicates how many errors have been found that need to be corrected before continuing to the next screen. The box’s content lists each of the errors with a detailed description of the error. Each of the error descriptions is linked to the input field: by clicking on an error description (or focusing it with a keyboard and pressing the `Enter` key), the respective input field is focused. In case of radio buttons, the first radio button is focused. Additionally to the error descriptions in the box, there is an error message next to each input field that requires correction, which also can be seen in Figure 4.16. These error messages are connected to their respective input field using the `aria-labelledby` attribute.

4.2.5 Tables

As opposed to the original Flight Booking Round Trip application, tables have column headlines (using the `<thead>` and `<th>` elements), which explain the meaning of their respective columns (e.g., “City” and “Code” in the tables of the first screen). This was done in order to avoid confusion when the meaning of the columns cannot be derived from the context. Additionally, zebra striped coloring was used for table rows, i.e., alternating background colors (white and light-gray) are used for table rows to distinguish them better from each other. Moreover, a gray line separates table rows. Each table also contains a table caption (e.g., “Select origin”) and summary (e.g., “Available origin cities and their details”) so that screen reader users quickly can recognize their meaning when skimming through the Web page.

4.2.6 Stacked Tables

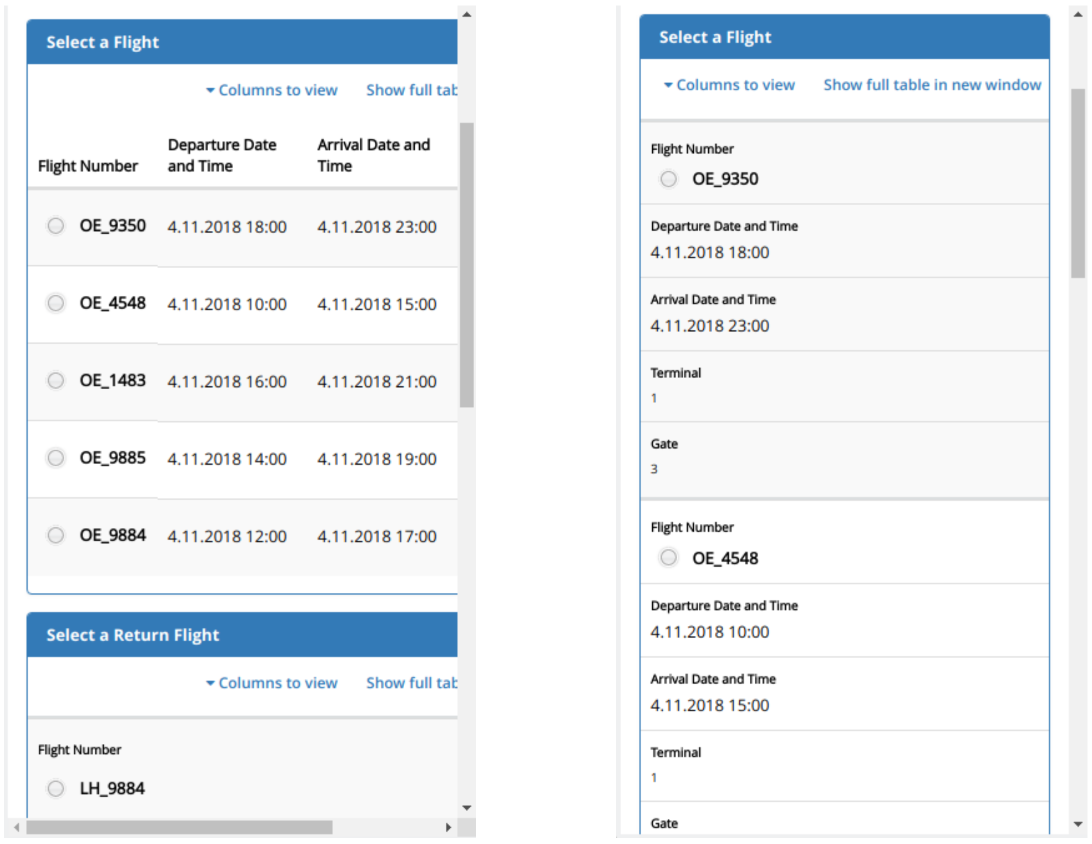
The “table to stacked table” widget replacement has been introduced after the user study in order to avoid horizontal scrolling in tables with many columns on smaller screens. While this form of displaying tabular data increases the amount of vertical scrolling that is needed (since the columns are displayed below each other), it avoids the more critical accessibility issue of having to scroll both vertically and horizontally at the same time.

Compare the two fictional examples of the flight selection table in Figure 4.17, where three new columns were introduced.

Both tables are viewed on a screen with a resolution of 375x667 pixels. On the left side, the table is shown in a typical tabular form, while on the right side, the table is shown as a stacked table. The advantage of the typical tabular form is that more table rows can be displayed on the screen. This comes at the disadvantage that scrolling is needed in both directions to view the whole Web page. With the stacked table solution, only two of the five rows in the example can be shown on the screen, but it does not require to scroll in two directions.

There is a trade-off between avoiding two scrolling axis and the amount of vertical scrolling. Since the impact of two scrolling axis can increase reading effort 40 to 100 times [CCGV08], it was decided to use stacked tables for small screens when displaying the columns in typical tabular form would introduce horizontal scrolling. There is also the possibility to show the whole table in a new window (by using the link at the top of the table), hence the user can decide for him- or herself which display form best suits his or her needs. By selecting fewer columns to view, the user can also reduce the amount of vertical scrolling as needed.

Additionally to the greater amount of vertical scrolling, stacked tables can also lead to less clarity in the arrangement. With tabular data, one can compare directly the values of a column for different rows (e.g., the departure date and time). This is not as easy anymore with stacked tables. Hence, it was decided to show stacked tables only if there are many columns that would lead to horizontal scrolling on smaller screens.



Select a Flight

▾ Columns to view   Show full tab

	Flight Number	Departure Date and Time	Arrival Date and Time
○	OE_9350	4.11.2018 18:00	4.11.2018 23:00
○	OE_4548	4.11.2018 10:00	4.11.2018 15:00
○	OE_1483	4.11.2018 16:00	4.11.2018 21:00
○	OE_9885	4.11.2018 14:00	4.11.2018 19:00
○	OE_9884	4.11.2018 12:00	4.11.2018 17:00

Select a Return Flight

▾ Columns to view   Show full tab

Flight Number
○ LH_9884

(a)

Select a Flight

▾ Columns to view   Show full table in new window

Flight Number
○ OE_9350

Departure Date and Time
4.11.2018 18:00

Arrival Date and Time
4.11.2018 23:00

Terminal
1

Gate
3

Flight Number
○ OE_4548

Departure Date and Time
4.11.2018 10:00

Arrival Date and Time
4.11.2018 15:00

Terminal
1

Gate

(b)

Figure 4.17: Showing tabular data in typical form (a) versus stacked form (b) on a screen resolution of 375x667 pixels

4.2.7 Dropdowns

Another widget replacement, “radio buttons to dropdown”, was introduced after the user study while analyzing the limitations of design-time approaches in GUI generation with regard to accessibility for low-vision disabilities [RTKP18]. In case there are more than six options in a radio button group, the radio buttons are replaced by a searchable dropdown (also called combo box). This noticeably reduces vertical scrolling. An example of a combo box is given in Figure 4.18, where the user has already limited the results shown in the dropdown by typing “an” into the input field. The input field text is highlighted in the results.

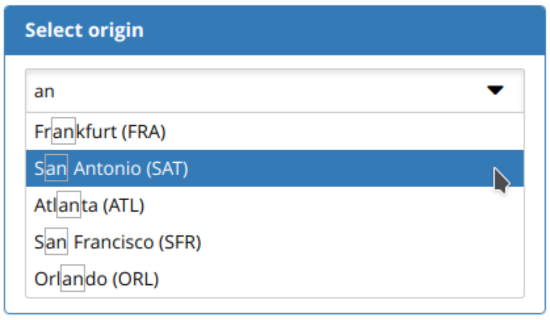


Figure 4.18: Combo box as widget replacement for radio button groups on small screens

4.2.8 Navigation

In the first version of the proof-of-concept application, a navigation was available, too, in order to demonstrate how an accessible navigation could be provided for applications that involve various steps in a process like most of the applications from Chapter 3 generated by UCP. Figure 4.19 illustrates the navigation in the first version of the proof-of-concept application.

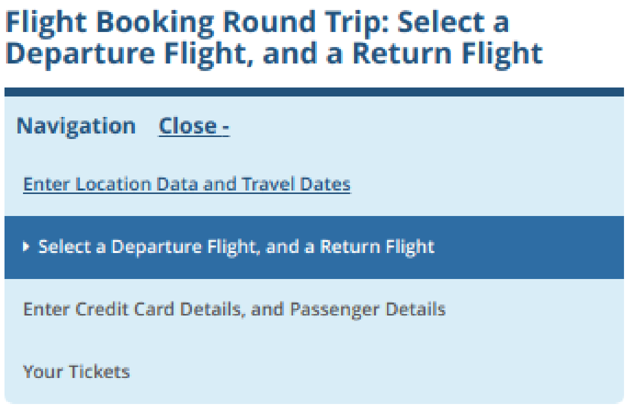

Flight Booking Round Trip: Select a Departure Flight, and a Return Flight
Navigation
Close -
Enter Location Data and Travel Dates
Select a Departure Flight, and a Return Flight
Enter Credit Card Details, and Passenger Details
Your Tickets

Figure 4.19: Navigation as given in the first version of the proof-of-concept application

The navigation makes it easy for the user to see how many steps there are in the process and how far the user has already progressed. Previous steps in the process are marked up with a hyperlink referencing to the respective screen. By clicking on the link, the respective Web page opens prefilled with the user's inputs. In this way, the user can always go back and check the data he or she has put in. The current screen in the process is indicated by a dark blue background, white text and an arrow symbol in the navigation. Subsequent steps are listed, but not clickable.

Next to the headline "Navigation" there is a button to open or close the navigation, Figure 4.20 shows the navigation in its closed state. The ability to close the navigation helps the user to save space on the screen when he or she already knows about the process steps, so that the user can focus on the actual contents of the current screen and is not distracted by the additional information. The information about whether the navigation is opened or closed is stored into a cookie so that its state is propagated among all the screens. Additionally, there is an invisible skip link (using the anchor attribute `href="#main"`) that allows screen readers to skip the navigation and move on with reading the main content of the application.

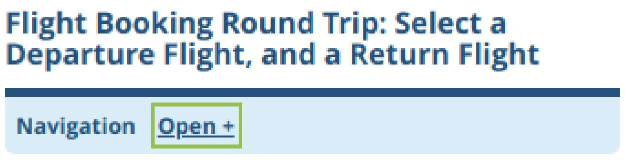

Flight Booking Round Trip: Select a Departure Flight, and a Return Flight
Navigation
Open +

Figure 4.20: Navigation in closed state as given in the first version of the proof-of-concept application

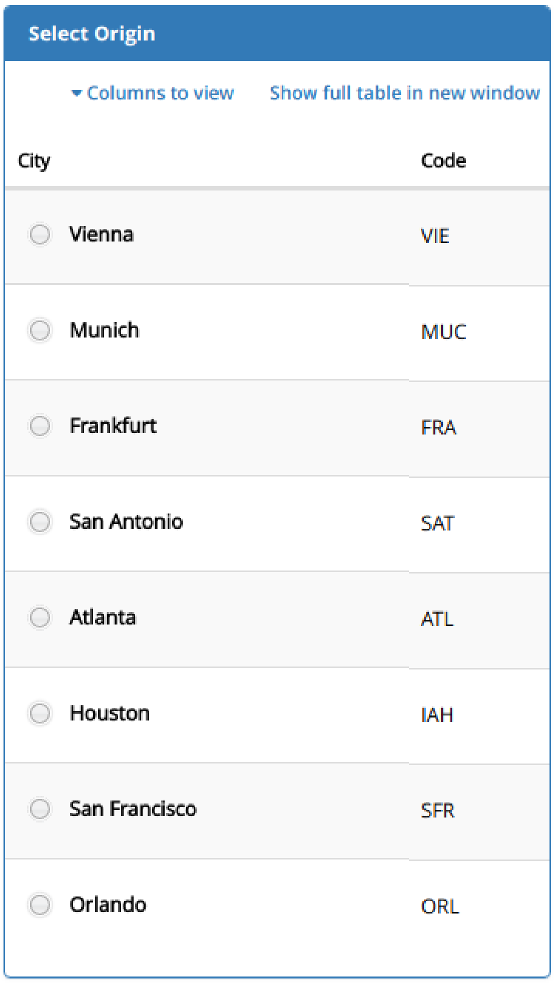
Since it became clear early on that implementing such a process navigation was not technically feasible in UCP without major enhancements because there is no distinction between process-like applications (like Flight Booking Round Trip) or applications involving a typical Website navigation (e.g., the sidebar navigation in Accommodation Booking), the navigation was excluded in later versions of the proof-of-concept application.

4.3 User Study

Before actually implementing these improvements in UCP, a user study was performed comparing the proof-of-concept application with the original Flight Booking Round Trip application generated by UCP.

The version used for the user study did not yet include the widget replacements "radio buttons to dropdown" or "table to stacked table". Moreover, the navigation was left out. The "table to stacked table" widget replacement was introduced to the proof-of-concept application at a later stage, since further research showed that this was a solution to prevent horizontal scrolling for more complex tables with many columns. Each table within the proof-of-concept application only has two columns, which is why that issue did not become apparent at first. The "radio button to dropdown" widget replacement got introduced during a research study that explored how the combination of design-time GUI generation and RWD can improve low-vision accessibility [RTKP18].

Figure 4.21 shows the version of tables that was used for the user study. Section 4.4.1 explains the differences between the implementation in UCP and the proof-of-concept application.



Select Origin

Columns to view   Show full table in new window

City	Code
○ Vienna	VIE
○ Munich	MUC
○ Frankfurt	FRA
○ San Antonio	SAT
○ Atlanta	ATL
○ Houston	IAH
○ San Francisco	SFR
○ Orlando	ORL

Figure 4.21: Version of a table as used during the user study

4.3.1 Setting and Process

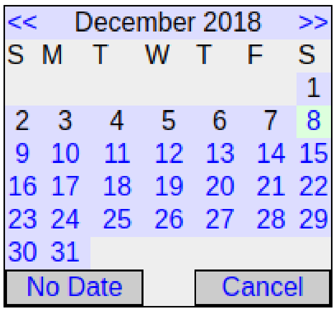
The user study was performed by students of the TU Wien as part of a user interface seminar in December 2017 and January 2018. 51 participants (17 female, 34 male; age between 13 and 60 years) were asked to perform the same predefined task in both the proof-of-concept application and the original Flight Booking Round Trip application. The participants were differently experienced in dealing with computers, some had regularly booked flights online before (but only on desktop computers), others did not. None of the participants were disabled.

The description of the task that the participants had to perform is presented in Figure 4.22. The birth year of Mr. Huber was originally defined to be 1970, but was set to 2017, since the date picker in the Original Flight Booking Round Trip application required a lot more time to navigate. Compare both date pickers in Figure 4.23. In the date picker of the proof-of-concept application, the user can navigate between consecutive years using the double arrows, while the double arrows in the original Flight Booking Round Trip application only allow navigating between consecutive months.

Imagine you are somewhere abroad, and your boss Mr. Huber tells you to book a flight ticket for his wife as quickly as possible.

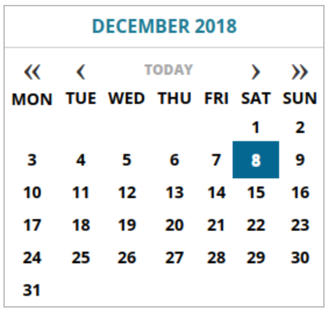
Book a flight from **Vienna** to **Munich** on Monday next week (10:00), with return on Friday (18:00), for Mrs. **Anna Huber** (born on **March 15, 2017**). Pay for it using her husband's (**Max Huber**) **VISA** credit card with the number **1258 8569 7532 1569** (CVC: **354**) and the expiration date **12/19**.

Figure 4.22: Task description used in the user study



S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

(a) Original Flight Booking Round Trip application



MON	TUE	WED	THU	FRI	SAT	SUN
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

(b) Proof-of-concept-application

Figure 4.23: Comparison of date pickers in the applications used for the user study

The user interfaces of the proof-of-concept application and the original Flight Booking Round Trip application were displayed consecutively on a smartphone, whose position was fixed on a table. Participants were asked to use only one hand for navigating through the applications. Participants were split into two groups, and the order in which the applications were presented to the participants differed between the groups (i.e., one group started with performing the task on the proof-of-concept application first, and then continued with the original Flight Booking Round Trip application, and vice versa). Additionally, the radio button options were presented in a random order, differing between both applications.

A video camera filmed the smartphone screen and the participants' hands while the task was being performed. The amount of time a participant has spent on each screen was analyzed afterwards. Page loading times and the time a participant spent using the software keyboard was excluded. After the participants had performed the task in both applications, they were asked to fill out a questionnaire asking for subjective impressions regarding both GUIs. Figure 4.24 lists all the questions being asked in the questionnaire.

1. Which interface is visually more attractive?
2. Which interface makes interaction more intuitive?
3. Which interface makes it easier to figure out what to do?
4. Which interface makes it clearer how to use it?
5. Which interface demands less time from you?
6. Which interface is easier to handle errors on?
7. Overall, which interface would you use to book flights?

Figure 4.24: Task description used in the user study

4.3.2 Results

When considering the subjective impressions the participants had regarding both GUIs, the proof-of-concept application clearly was found visually more attractive (73 % preferred it over the original Flight Booking Round Trip application) and participants would prefer to use it when booking flights (58 %). No clear preference was shown with regard to how intuitive, clear, or easy the GUI was, probably because both GUIs were too similar (several participants had mentioned that). 34 % of the participants found the proof-of-concept application easier to handle errors on, while 44 % thought that both GUI were similarly easy in that regard.

Many participants claimed that the proof-of-concept application demanded less time from them (42 % said so). However, the time spent on each screen measured from the video tells a different story. On average, in the proof of concept application, participants spent almost 13 seconds more on the first screen and 6 seconds more on the second screen, than in the original Flight Booking Round Trip application. A reason for this is likely that the participants had to scroll a lot more in the proof-of-concept application, because the spaces between elements were much larger. Some participants that liked the original Flight Booking Round Trip GUI more also mentioned in the questionnaire that the reason for this was that it required less scrolling. Additionally, the font size in the original Flight Booking Round Trip was larger, which made it easier to read the text.

Some participants preferred the proof-of-concept application, since they could click on the label to select a radio button option, while in the original Flight Booking Round Trip application they had to click on the radio button itself, which was a much smaller target region. Some participants seemed to make less errors in the proof-of-concept application because of that.

The date picker in the proof-of-concept application was preferred by some participants, since selecting a date seemed to be easier. However, other participants mentioned that they were a little bit confused because the date picker did not open when focusing the date input field (only when clicking on the date picker icon). Some participants preferred that the proof-of-concept application showed the required date format as a hint below the input field.

Overall, the similarity between the two user interfaces was emphasized by several participants.

4.3.3 Lessons Learned

Even though the user study did not include disabled participants, some lessons have been learned from it regarding accessibility. The major findings regarding accessibility were:

- Since the click targets of the radio buttons (due to the labels) are larger, it is less likely to make errors and easier to select an option.
- The original Flight Booking Round Trip required far less scrolling due to less spacing between elements. Users needed more time and more scrolling to finish their tasks in the proof-of-concept application.
- There is a trade-off between the spacing between elements and the amount of scrolling needed. A larger space between elements increases click targets and hence can lead to less errors. However, a higher amount of scrolling increases overall task performing time.
- Although the spaces between elements are larger in the proof-of-concept application, the font size is slightly smaller than in the original Flight Booking Round Trip application. The larger font size in the original Flight Booking Round Trip makes it easier to read the text.
- Although the calender was perceived as being superior to the old version, users wished it would open immediately when clicking into the date input field.
- Even non-disabled people would like to have input hints (like showing the required date format for a date input field), even though they likely can correct their errors much quicker than disabled people.
- The fact that the birth year of Mr. Huber in the task description had to be changed due to limitations of the date picker in the original Flight Booking Round Trip application shows that special care needs to be taken when selecting GUI component libraries. Since the original date picker would require a lot more clicks, especially

people with motor disabilities would face issues and would require a lot more time to select the correct date.

These learnings have been used for improving the proof-of-concept application and implementing the accessibility improvements in UCP.

4.4 Implementation of Accessibility Improvements in UCP

After the evaluation of the proof-of-concept application, the accessibility improvements were implemented in UCP. Some features have slightly changed compared to the proof-of-concept application due to the results from the user study and further research. The implementation was bounded to two modules within UCP that are responsible for generating the FUI (i.e., GUI source code). Some new technologies have been introduced into UCP for inclusion of some run-time behavior and adaptations.

4.4.1 Differences to the Proof-of-Concept Application

Compared to the proof-of-concept application evaluated in the user study, some features and stylings have been changed based on findings in the user study and further research.

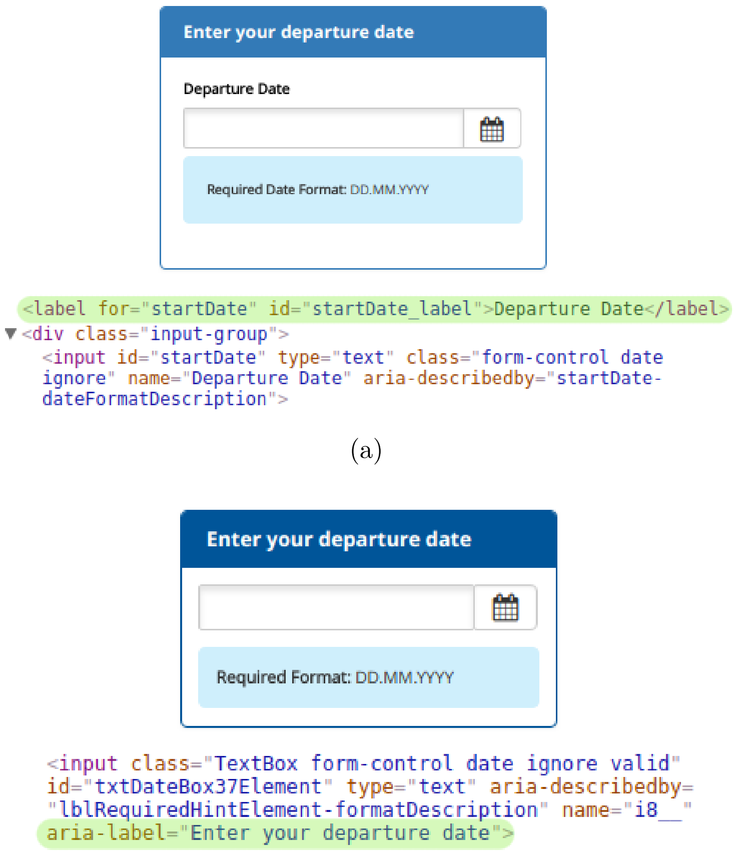
To reduce scrolling and eliminate unnecessarily duplicated information, some labels (e.g., the date picker labels “Departure Date” and “Return Date”) have been removed again from the visible screen and made implicit using the ARIA attribute `aria-label`, which is demonstrated in Figure 4.25. This ensures that the label text is still available for screen readers, while being visually hidden. Since the information is already shown in the panel title, it does not make sense to show it twice (both the panel title and as a label for the input field directly below). The screen reader still needs the information in case the user jumps from one input field to the next using the `Tab` key on the keyboard, in which case the panel title would not be read by the screen reader.

The font size was increased and the spaces between GUI components was reduced in order to make the text more readable while reducing the amount of scrolling at the same time (see Figure 4.26 for comparison).

Additionally, during manual accessibility evaluation, it was noticed that the font size could not be customized via browser settings, since the font size of the `<body>` tag was given in pixels (which was introduced by the Bootstrap framework). This was solved by setting the font size of the `<body>` text to a relative measure (`1.1em`).

Furthermore, the color contrast has been enhanced for headings and buttons to suffice the level AAA contrast requirement.

The button sizes (e.g., for submitting or canceling a form) have been increased to have a minimum width of 120 pixels on large screens in order to increase the click target. The


Enter your departure date
Departure Date
Required Date Format: DD.MM.YYYY

```
<label for="startDate" id="startDate_label">Departure Date</label>
▼<div class="input-group">
   <input id="startDate" type="text" class="form-control date
   ignore" name="Departure Date" aria-describedby="startDate-
   dateFormatDescription">
```

(a)

Enter your departure date
Required Format: DD.MM.YYYY

```
<input class="TextBox form-control date ignore valid"
id="txtDateBox37Element" type="text" aria-describedby=
"lblRequiredHintElement-formatDescription" name="i8__"
aria-label="Enter your departure date">
```

(b)

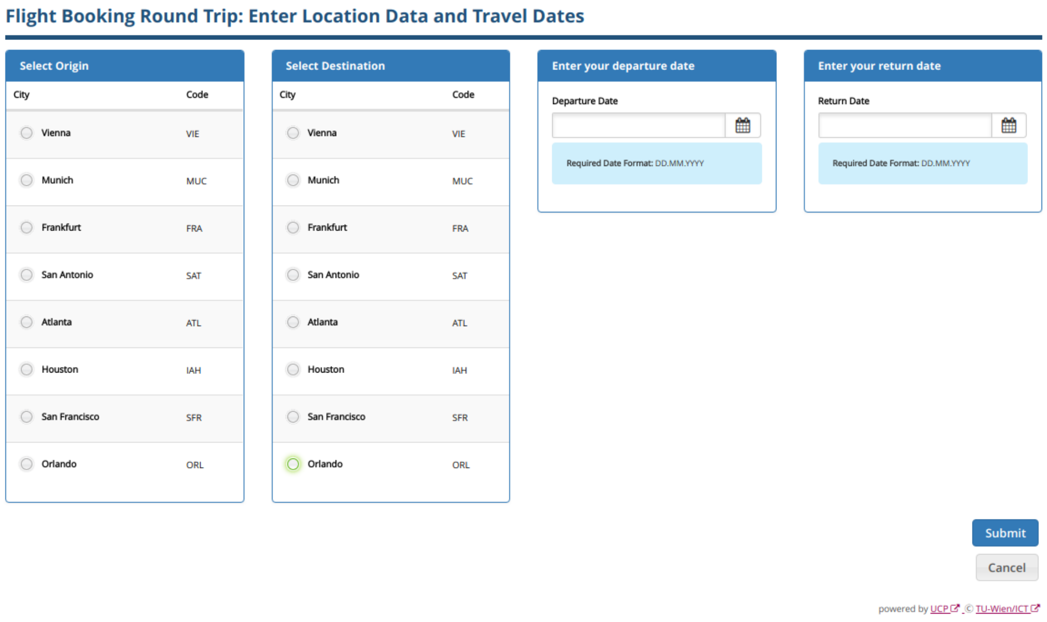
Figure 4.25: Comparison of explicit (a) and implicit (b) labels

table summary has been excluded since it is not automatically derivable from the model within UCP. However, the table's caption is, in fact, derivable from the model by using the description of the Closed Question within the discourse model ("Select Origin"), which is also used for the panel title.

The tab view component has been made accessible. This has not been covered by the proof-of-concept application, since there was no use case in the original Flight Book Round Trip application that would have been suitable for a tab view.

Finally, the widget replacement "Radio Buttons to Dropdown" is not only performed for small screens with a width lower than 992 pixels, but in general, if there are more than six options. This was done in order reduce mouse movements also on large screens. It could be possible in the future to set different thresholds (i.e., number of options) for this widget replacement on desktop computers, tablets and smartphones using device specifications.

The date picker component was not changed to open immediately when clicking into



Flight Booking Round Trip: Enter Location Data and Travel Dates

Select Origin

City	Code
Vienna	VIE
Munich	MUC
Frankfurt	FRA
San Antonio	SAT
Atlanta	ATL
Houston	IAH
San Francisco	SFR
Orlando	ORL

Select Destination

City	Code
Vienna	VIE
Munich	MUC
Frankfurt	FRA
San Antonio	SAT
Atlanta	ATL
Houston	IAH
San Francisco	SFR
Orlando	ORL

Enter your departure date

Departure Date

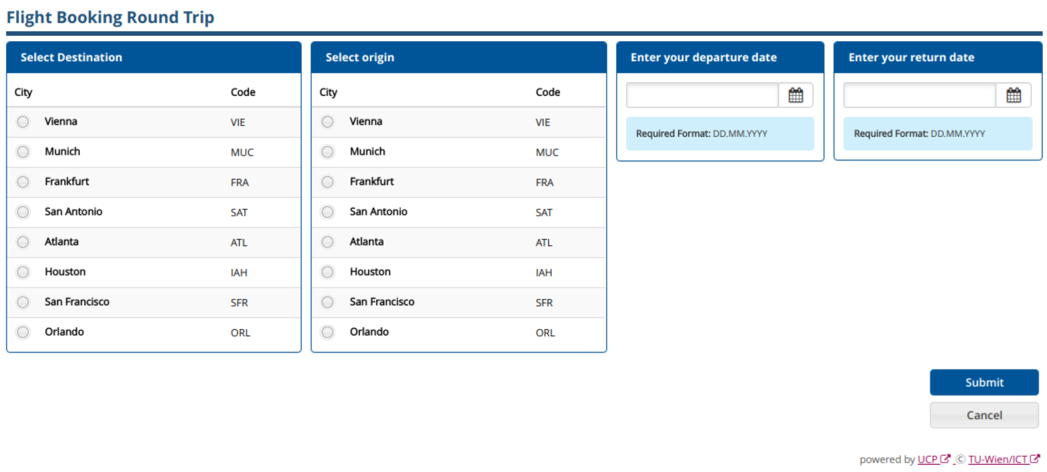
Required Date Format: DD.MM.YYYY

Enter your return date

Return Date

Required Date Format: DD.MM.YYYY

(a)



Flight Booking Round Trip

Select Destination

City	Code
Vienna	VIE
Munich	MUC
Frankfurt	FRA
San Antonio	SAT
Atlanta	ATL
Houston	IAH
San Francisco	SFR
Orlando	ORL

Select origin

City	Code
Vienna	VIE
Munich	MUC
Frankfurt	FRA
San Antonio	SAT
Atlanta	ATL
Houston	IAH
San Francisco	SFR
Orlando	ORL

Enter your departure date

Required Format: DD.MM.YYYY

Enter your return date

Required Format: DD.MM.YYYY

(b)

Figure 4.26: Comparison of the font sizes and spacings between the proof-of-concept application (a) and the implementation in UCP (b)

the date input field, since it should be possible to either put in the date manually or to select it in the date picker. Both cannot be enabled at the same time, since the date picker would need to be synced with the date input field as soon as the user starts typing. The date picker would jump back and forth while the user is typing, which would be confusing. It would also be unclear how to switch focus between the input field and the picker. Additionally, the success criteria “3.2.1 On Focus” could be interpreted in a way that would make the date picker component fail. It requires that, when focusing a component, the focus is not changed to another component. That would be the case for the date picker when focusing the date input field (since it would immediately switch focus to the date picker).

4.4.2 Scope and Boundaries of Implementation

The implementation within UCP was limited to the two modules responsible for generating the FUI (`org.ontoucp.structuralui.code.generator` and `org.ontoucp.structuralui.code.generator.html`). At this point in the generation process, the Structural UI Model, i.e., the abstract representation of the GUI that already contains information about the GUI’s structure and content, is already available. This implies that all the information needed for implementing accessibility features in the FUI needs to be available at this point.

The Structural UI Model did not yet support some of the features, namely input format hints, table headers, connections between labels and their input fields, as well as widget replacements. The missing model components have been provided by UCP developers. Since these additional components cannot yet be generated from the Communication Model, the (generated) Structural UI Model of the Flight Booking Round Trip application was modified manually to incorporate all the missing features. This extended model demonstrates how the model will have to look like so that accessible GUIs can be generated in the future. Based on this extended model, the code generation was implemented as part of this thesis.

The use case in which additional content is shown on the screen after a page reload, like it is the case in the Shopping application, has not been addressed. Although this poses an accessibility issue for blind users, as discussed in Section 3.3.1, fixing it would require a substantial modification of the framework’s runtime framework to load contents without requiring a page reload. This was out of the scope of this thesis.

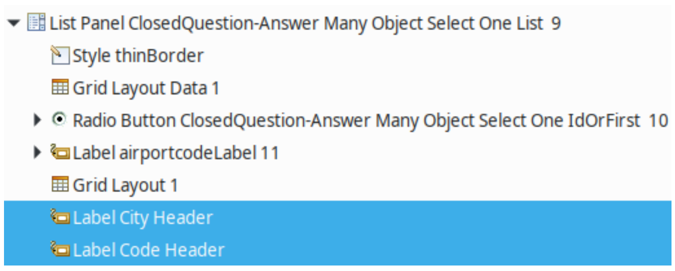
Since there is currently no distinction between process-like applications and applications involving a conventional Website navigation, the navigation discussed in Section 4.2.8 has not been implemented in UCP.

4.4.3 Changes to the Structural UI Model

As mentioned above, the Structural UI Model had to be modified manually in order to be able to implement the code generation of the proposed accessibility features, since UCP did not support some of these features at that time. Actually, two manually modified Structural UI Models have been created based on the (generated) Structural UI Model of the Flight Booking Round Trip application. The first one includes the widget replacements “Radio Buttons to Dropdown” and “Table to Stacked Table”. The second Structural UI Model was generated using a device specification for smartphone resolutions and includes a tab component, which is replaced by an accordion at a defined media breakpoint. These Structural UI Models demonstrate how the models should look like when they would be generated by UCP in the future. The remainder of this section explains the manual changes in the Structural UI Model for each accessibility feature which could not be implemented using the generated Structural UI Model as is.

Table Headers

To show the headers of table columns in tables and stacked tables, a way of adding labels to the list widget has been introduced by UCP developers. A list widget can contain a list of these labels. Figure 4.27 shows the header labels within the list panel for selecting the departure airport. A table's column header text is represented by the "Text" property of the header label.



```
▾ List Panel ClosedQuestion-Answer Many Object Select One List 9
    Style thinBorder
    Grid Layout Data 1
  ▸ Radio Button ClosedQuestion-Answer Many Object Select One IdOrFirst 10
  ▸ Label airportcodeLabel 11
    Grid Layout 1
    Label City Header
    Label Code Header
```

Figure 4.27: Table header label (highlighted) within the list panel component for selecting the departure airport

Connections between Labels and Input Fields

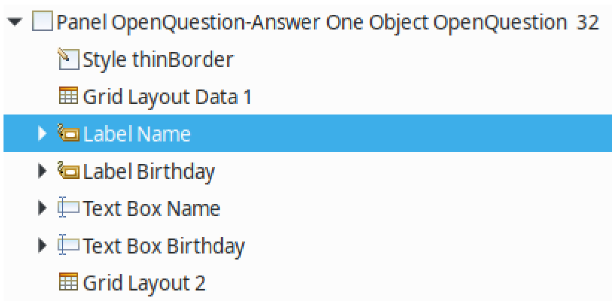
To be able to connect a label with its respective input field by setting the label's `for` attribute, a property called "For" was added to the label widget, which specifies a link to the corresponding input widget. Figure 4.28 shows the representation of the panel for entering personal data in the Structural UI Model, which contains two labels and two text input fields. Below the model, the "Name" label's properties are shown, in which the "For" property has been set to the input widget "Text Box Name".

Input Hints

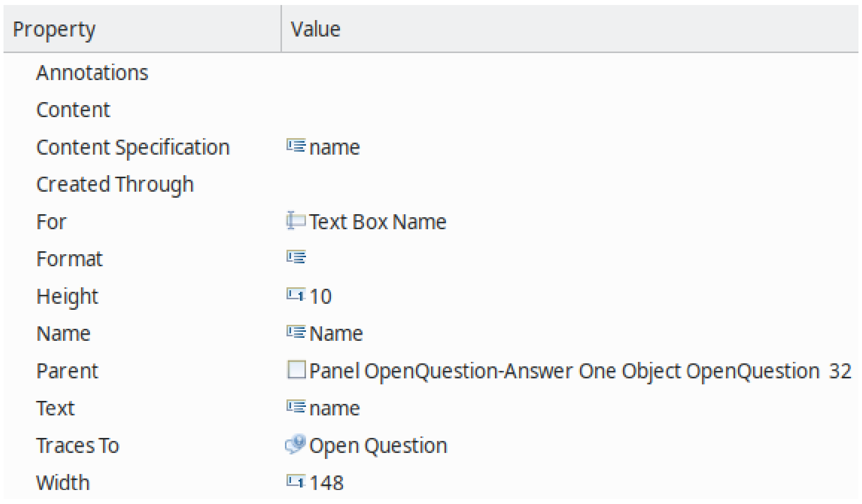
Input hints are represented by label widgets that have a special style (i.e., "formatHint") associated with them. The are connected to their respective input field via the "For" attribute. The format is specified by the label's "Format" property and its "Text" property holds the hint's description. Figure 4.29 shows the input hint's representation in the Structural UI Model and its properties.

Widget Replacements

For widget replacements, two new components have been added by UCP developers: the Display Alternative and its child component, the Alternative. A Display Alternative



- Panel OpenQuestion-Answer One Object OpenQuestion 32
  - Style thinBorder
  - Grid Layout Data 1
  - Label Name
  - Label Birthday
  - Text Box Name
  - Text Box Birthday
  - Grid Layout 2



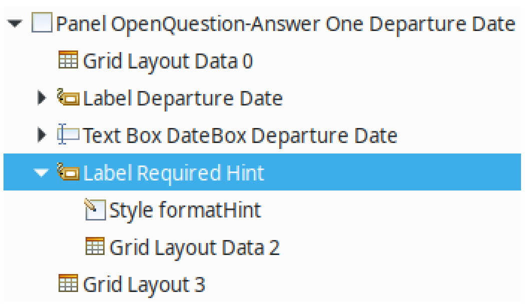
Property	Value
Annotations	
Content	
Content Specification	name
Created Through	
For	Text Box Name
Format	
Height	10
Name	Name
Parent	Panel OpenQuestion-Answer One Object OpenQuestion 32
Text	name
Traces To	Open Question
Width	148

Figure 4.28: Connection between the label “Label Name” and the input widget “Text Box Name” via the label’s “For” property

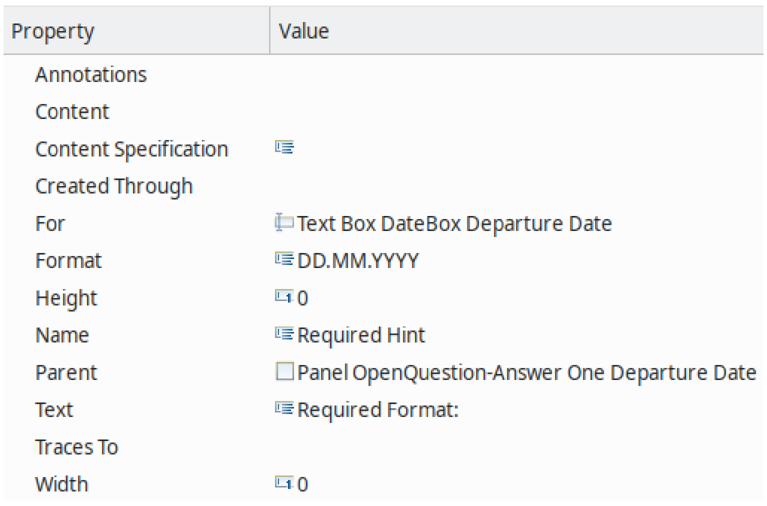
wraps multiple Alternative elements, which contain all GUI components that should be shown under a certain display condition. The Display Alternative’s intention is to show only one Alternative at any given time. To do this, mutually exclusive conditions are assigned to properties of each Alternative. Currently, to model the widget replacements introduced in Section 4.1.5, two conditions have been added to the Alternative via properties. In the future, more conditions could be added as needed.

The first condition is the range of screen width for which the Alternative is shown, given by the “Minimum Screen Width” and “Maximum Screen Width” properties. These are the media breakpoints at which the Display Alternative switches from one Alternative to another. When generating the Structural UI Model in the future, it has to be made sure that the complete range of screen widths is covered, otherwise no Alternative would be shown for non-covered ranges.

The second condition is related to the amount of options shown in an Alternative. This



- Panel OpenQuestion-Answer One Departure Date
  - Grid Layout Data 0
  - Label Departure Date
  - Text Box DateBox Departure Date
  - Label Required Hint
    - Style formatHint
    - Grid Layout Data 2
  - Grid Layout 3

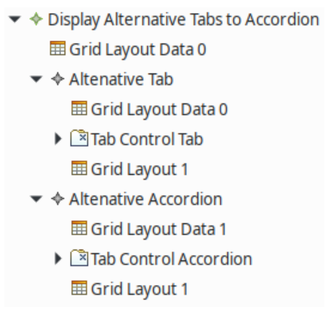


Property	Value
Annotations	
Content	
Content Specification	
Created Through	
For	Text Box DateBox Departure Date
Format	DD.MM.YYYY
Height	0
Name	Required Hint
Parent	Panel OpenQuestion-Answer One Departure Date
Text	Required Format:
Traces To	
Width	0

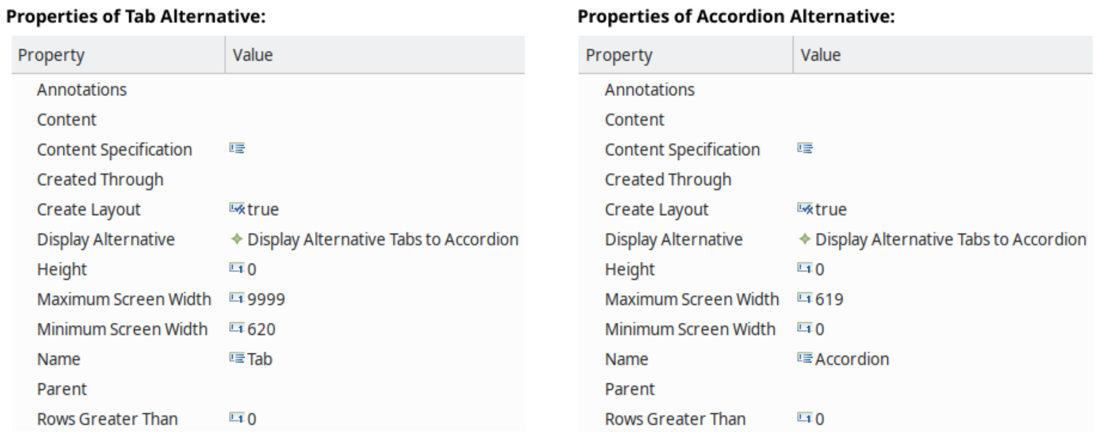
Figure 4.29: Representation of an input hint for the departure date

is currently only used for radio to dropdown widget replacements, but could also be extended to other widget replacements (e.g., the number of tabs in the tab to accordion widget replacement). The condition is defined by the “Rows Greater Than” property of the Alternative which indicates that it should be shown only if it contains more options than the number given in the property. The upper limit of rows for which the Alternative will be shown is given by the “Rows Greater Than” property of its sibling Alternatives. For example, imagine a Display Alternative contains two Alternatives, one which shows a list of radio buttons and the other showing a combo box. The radio button list Alternative has set a “Rows Greater Than” of 0, and the combo box Alternative has the property set to 6. The radio button list will be shown when it contains one to six options, while the combo box will be shown if there are more than six options.

Figure 4.30 shows the model of the *“Tabs to Accordion”* widget replacement. The Display Alternative contains two Alternatives, one for showing tabs and one for showing accordions. Both Alternatives contain the tab component. Its display type (i.e., tab or accordion view) is determined by a style applied to it (not shown in Figure 4.30). The



- Display Alternative Tabs to Accordion
  - Grid Layout Data 0
  - Altenative Tab
    - Grid Layout Data 0
    - Tab Control Tab
    - Grid Layout 1
  - Altenative Accordion
    - Grid Layout Data 1
    - Tab Control Accordion
    - Grid Layout 1



Properties of Tab Alternative:

Property	Value
Annotations	
Content	
Content Specification	
Created Through	
Create Layout	true
Display Alternative	Display Alternative Tabs to Accordion
Height	0
Maximum Screen Width	9999
Minimum Screen Width	620
Name	Tab
Parent	
Rows Greater Than	0

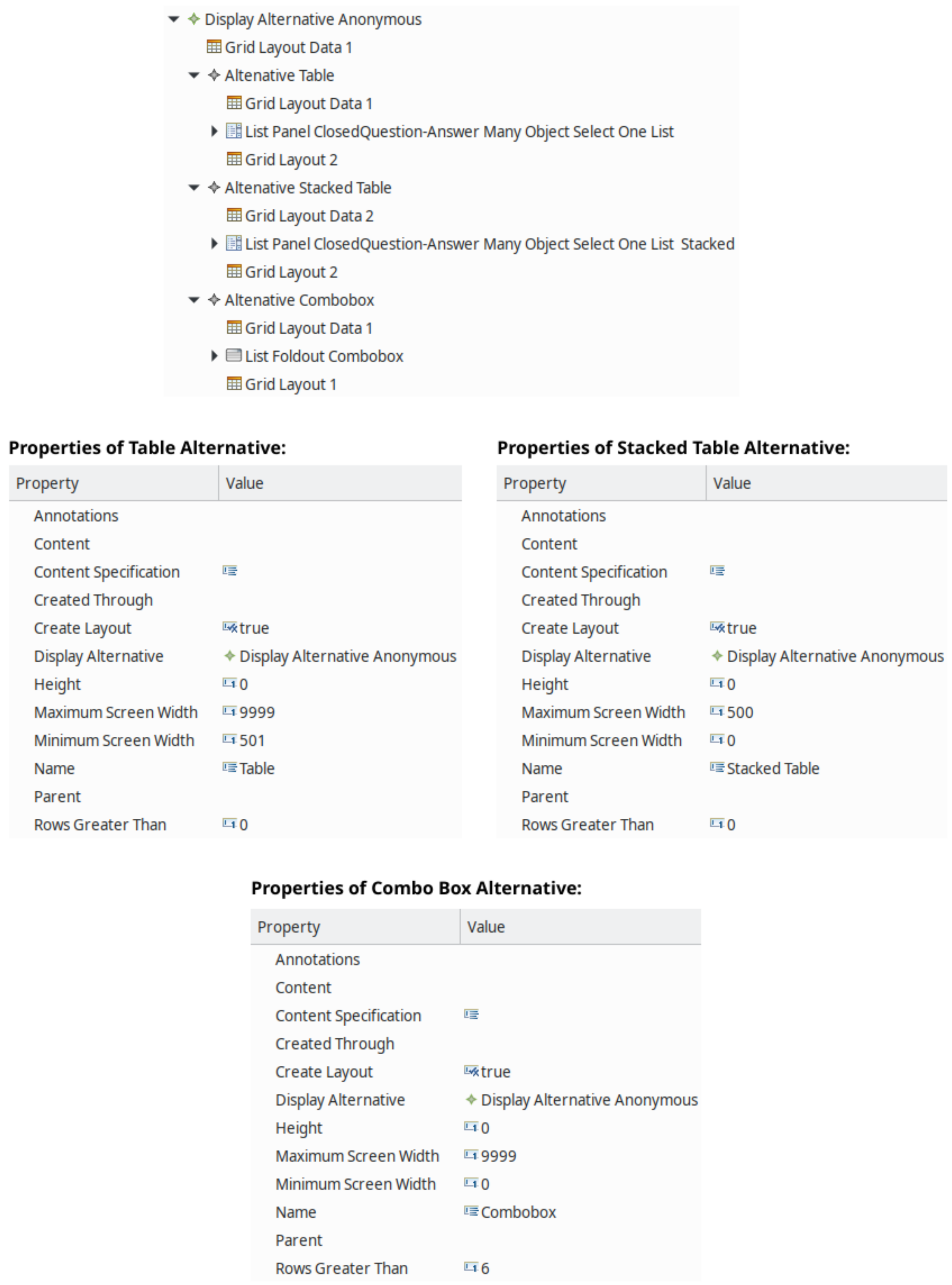
Properties of Accordion Alternative:

Property	Value
Annotations	
Content	
Content Specification	
Created Through	
Create Layout	true
Display Alternative	Display Alternative Tabs to Accordion
Height	0
Maximum Screen Width	619
Minimum Screen Width	0
Name	Accordion
Parent	
Rows Greater Than	0

Figure 4.30: Example model and properties for the “Tabs to Accordion” widget replacement

tab Alternative is shown when the screen width is greater than or equal to 620 pixels. The width of 620 pixels is exactly the width for which the tabs still can be viewed in one line. The accordion Alternative is shown when the screen width is smaller than 620 pixels. The “Rows Greater Than” property is set to 0 for both Alternatives, since it should not affect the decision of which Alternative to show.

Figure 4.31 illustrates the *“Radio Buttons to Dropdown”* and *“Table to Stacked Table”* widget replacements. The Display Alternative contains three Alternatives: the first one shows the data as a typical table, the second one shows it as a stacked table, and the third one shows it as a combo box. Both conditions, the screen width and the number of options, determine which Alternative should be shown. If there are more than six options for the Display Alternative (as there are, for example, when selecting the origin and destination airports), the combo box Alternative is shown regardless of the screen width. If there are fewer than six options (e.g. when selecting flights and return flights), the screen width determines which kind of table is shown. For a screen width of up to 550 pixels, the stacked table is displayed, otherwise the typical table is shown.



- Display Alternative Anonymous
  - Grid Layout Data 1
  - Altenative Table
    - Grid Layout Data 1
    - List Panel ClosedQuestion-Answer Many Object Select One List
    - Grid Layout 2
  - Altenative Stacked Table
    - Grid Layout Data 2
    - List Panel ClosedQuestion-Answer Many Object Select One List Stacked
    - Grid Layout 2
  - Altenative Combobox
    - Grid Layout Data 1
    - List Foldout Combobox
    - Grid Layout 1

Properties of Table Alternative:

Property	Value
Annotations	
Content	
Content Specification	☰
Created Through	
Create Layout	true
Display Alternative	Display Alternative Anonymous
Height	0
Maximum Screen Width	9999
Minimum Screen Width	501
Name	Table
Parent	
Rows Greater Than	0

Properties of Stacked Table Alternative:

Property	Value
Annotations	
Content	
Content Specification	☰
Created Through	
Create Layout	true
Display Alternative	Display Alternative Anonymous
Height	0
Maximum Screen Width	500
Minimum Screen Width	0
Name	Stacked Table
Parent	
Rows Greater Than	0

Properties of Combo Box Alternative:

Property	Value
Annotations	
Content	
Content Specification	☰
Created Through	
Create Layout	true
Display Alternative	Display Alternative Anonymous
Height	0
Maximum Screen Width	9999
Minimum Screen Width	0
Name	Combobox
Parent	
Rows Greater Than	6

Figure 4.31: Example model and properties for the “Radio Buttons to Dropdown” and “Table to Stacked Table” widget replacements

4.4.4 Implementation

This subsection explains how the accessibility features were implemented in UCP based on the manually adapted Structural UI Models.

General Structure of the Screen

In UCP, the GUI is built from top to bottom, starting at the page (called "Screen") and moving down the Structural UI Model tree to generate all the components into a Velocity (`.vm`) template file, recursively. Additionally, a controller and service, responsible for loading the current Web page and all its resources, as well as a messages file are generated, and all resources (e.g., CSS and JavaScript files) are moved to folders from which they can be served.

This process was extended to copy all the new libraries introduced in Chapter 2 and to generate the new GUI components (namely `DisplayAlternative` and `Alternative`) and a new JavaScript file for adding runtime behavior. While in the original UCP framework, only an HTML file per screen was generated, now there are both an HTML and a JavaScript file generated per screen.

The generated Web page looks as follows: The HTML tag contains a `lang` attribute which is set to the US locale (`en-US`), determining the language of the page. Two JavaScript files are loaded, `modernizr.js` for feature detection and `require.js`, which then will load all the necessary JavaScript files for the page asynchronously. Modernizr detects, for example, which browser is being used and certain styles are then added by the Assets framework depending on the browser (which is needed for some older browsers to circumvent their lack of support of newer CSS and JavaScript features). The body of the HTML document contains a `<div>` element with the `container` class to wrap Bootstrap's grid layout. A `<noscript>` tag wraps a message shown to the user in case JavaScript was disabled to indicate that not all functionality might be available. The page is structured by three sections: the header, a main area, and a footer. The header contains the application's title, the footer contains hyperlinks to the UCP project Website and the Website of the Institute of Computer Technology at TU Wien, and the main section holds all the generated GUI components wrapped by a global `<form>` element.

Layout Mechanism for the Screen

The layout mechanism of UCP had to be changed from a table layout to Bootstrap's responsive grid layout. In the Structural UI Model, the layout is given by a grid layout: each container defines how many rows and columns it holds and the dimensions (width and height) of a cell (i.e., column width and row height). Each widget (including containers) defines in which column and row it is placed, how many rows and columns it spans, and its alignment within a cell (e.g., `NORTH_WEST` for a top left alignment).

A new utility class, `LayoutGridUtils`, takes care of transforming this grid layout into Bootstrap's 12-column-grid. To do this, the columns have to be translated to fit that format (i.e., they have to be spread among the 12 columns that make up a Web page's width). First, the width of each container is divided by 12 in order to get the width of one column. Then, the width of each grid layout column within the container is divided by the calculated column width. If there is a column span defined on a component, the calculated values for the corresponding cells are added. As a result, each component within a container gets assigned a number that determines the number of columns it spreads among the 12-column-grid of its parent container (e.g., 3 if the components width makes up a quarter of the container's width). This number is then used to construct the component's CSS class to determine its width (e.g., `col-md-3`). For now, only one type of grid classes (i.e., `col-md` for medium-sized screens) is used, which automatically adapts itself to smaller screens by expanding the widths of the columns to that the components will eventually reflow to a one-column-layout. This could be easily extended by specifying multiple widths (one for each device specification), so that the grid becomes more fine-grained.

Another new utility class, `PrimaryPanelUtils`, was introduced to distinguish bordered panels from other containers that contain no styling (which are only used for creating the layout). The bordered panels are determined by which element within the discourse the corresponding element in the Structural UI Model represents. In case it is an Open or Closed Question, an Informing or Accept element, its contents will be wrapped within Bootstrap's panel component. In case the element holds a label with a "heading" style, that label will be put into the panel's header.

Form Elements

The newly introduced utility class `FormUtils` takes care of deciding the styling, type and behavior of form elements.

A label widget's type and styling is defined by which styles are specified for it in the Structural UI Model and whether it has set its "For" property to an input widget:

1. Label widgets having the "heading" style are interpreted as headlines (typically as panel headlines) and are marked up with HTML `<h*>` tags.

2. Format hints are defined by the "formatHint" style and show not only the label widget's text, but also its format, which resembles the meaning of the data format for its corresponding input field. A format hint label widget also needs to have set its "For" property to an input widget, so that it can be connected to its corresponding input element via the `aria-describedby` attribute of the input element in HTML.

3. Label widgets which have their "For" property set but do not have a specified styling, are interpreted as true HTML `<label>` elements. They are connected to their input field by the HTML `for` attribute.

4. Finally, label widgets with none of the above mentioned stylings or an empty "For" attribute (or when it is set to a widget that is not an input widget), are interpreted as plain text.

Besides being connected to their format hints, input elements can also have an `aria-label` attribute, which is set to the panel headline, if no label widget was defined connected to the input widget. Using the panel headline as an input element's label is sensible in cases where there is only one input widget within the panel, as it is with the date picker in the Flight Booking Round Trip application for the departure date and the return date, respectively. Otherwise (i.e., if there is more than one input widget within a panel), the Structural UI Model should always provide a label widget along with the input widget.

Date input fields are indicated via the `date` CSS class that is detected by a JavaScript function on page load, which in turn initializes the date pickers, i.e., adding the button for opening the date picker as well as adding accessibility-related markup to the date input field. Figure 4.32 illustrates the input mask being used for indicating the required date format, so that the user knows how to provide the date by typing in if he or she chooses not to use the date picker. The date picker button uses a Font Awesome icon of a calendar, which can also be seen in Figure 4.32. It is augmented by a hidden text accessible to screen readers that says "Show calendar for [label name]". Figure 4.33 shows this for the date picker in UCP: The `<span>` element containing the class `sr-only` is hidden from the screen by setting its positioning to "absolute", its dimensions to 1x1 pixels and its margin to -1 pixel. Hence, the element is moved out of the screen, but still remains in the DOM to be accessible by screen readers. Additionally, the font icon wrapped in the `<span>` element with the `fontIcon` class is hidden from screen readers using the `aria-hidden` attribute, which is set to `true`.

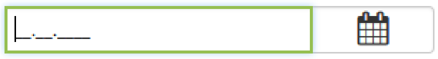
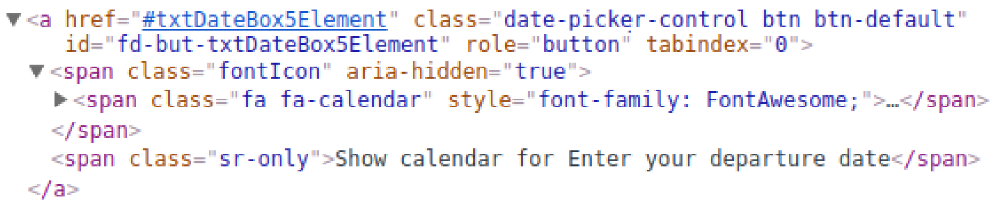


Figure 4.32: Input mask and Font Awesome icon used in the date picker component



```
▼<a href="#txtDateBox5Element" class="date-picker-control btn btn-default"
   id="fd-but-txtDateBox5Element" role="button" tabindex="0">
  ▼<span class="fontIcon" aria-hidden="true">
    ▶<span class="fa fa-calendar" style="font-family: FontAwesome;">…</span>
    </span>
    <span class="sr-only">Show calendar for Enter your departure date</span>
  </a>
```

Figure 4.33: Font icon on the link for the date picker in UCP

For radio buttons, their text has been turned into an HTML label, so that they are read by screen readers when the radio button is focused and the user can click on the text to select the option, hence increasing the click target.

Combo boxes are implemented using the Combobo library introduced in Chapter 2 and contain the HTML attribute `role="combobox"`, since there is no dedicated HTML element resembling a combo box. There is an `input` element which owns a container holding the available options by having the ARIA attribute `aria-owns`. Further ARIA attributes it contains are `aria-autocomplete="list"` and `aria-expanded`. The attribute `autocomplete="off"` prevents the browser from adding a list of suggestions (determined by the user's input history) to the input field while the user is typing, since these would overlap the actual options of the dropdown. The container holding the options has the role `listbox` and the options have the role `option`. JavaScript functions and listeners provided by the Combobo library make sure that ARIA attributes (like `aria-selected`) are being kept up-to-date while the user browses through and selects options. Additionally, filtering options as the user types into the input field and highlighting parts of the options that match the input, are done via the library.

Tab Controls and List Widgets

Tab controls can have two appearances, the typical tab view (indicated by the `tab` CSS class), and the accordion view (indicated by the `accordion` CSS class). In HTML, the tabs are represented by an unordered list of hyperlinks referencing to their corresponding tab contents. Each link has the ARIA attributes `aria-setsize` and `aria-posinset` to indicate how many tabs there are and the respective tab's position. Further accessibility-relevant markup is added via a JavaScript function on page load.

List widgets show their list elements within an HTML table. The table can also have two appearances, the typical table view and the stacked table view (indicated by the `stacked` CSS class). Table headers are generated through the header labels (i.e., the special kinds of labels added manually to the Structural UI Model). To show the column headers also in the stacked table version, the `data-title` attribute of a table cell is also set to the column header text. The possibility for a user to select which columns he or she wants to be displayed could not be implemented at this point, since it was not clear at the time how columns that always need to be shown (e.g., because they are required for user interaction or are necessary for the user to make sense of the other columns) could be determined during model generation.

Runtime Capabilities

A few runtime capabilities have been added to UCP via JavaScript. For this, a JavaScript file (`<screenID>.js`) is generated for each screen which loads the relevant dependencies that are needed for the current Web page. It is defined as a module that includes all the runtime logic for the current screen as well as various dependencies needed to provide that logic. The newly introduced utility class `RuntimeBehaviorUtils` determines, which dependencies need to be included for the respective screen. Some dependencies (like jQuery, Bootstrap and the Assets Framework) are always loaded since they are needed for initializing the Web page. Others, like scripts used for initializing forms, tables, or tab controls are only loaded when the page contains corresponding elements to reduce page loading time. Also, the contents of `<screenID>.js` are determined by which elements are actually on the page. This is done by recursively traversing the tree of components from the Structural UI Model using a function in the `RuntimeBehaviorUtils`. `<screenID>.js` is loaded using Require.js at page load. To pick the correct file (since each screen has its own JavaScipt file), the `<body>` tag of the HTML file contains a `data-screen` property, which is set to the unique `<screenID>` for the respective screen.

At page load, several components that require JavaScript for their functionality are initialized in `<screenID>.js`. For example, the date pickers are created by specifying the date format, their labels and an input mask corresponding to the date format. Additionally, all the relevant rules for the form checking are created. This is done by taking the `format` property of the screen's input widgets into account. For example, the `format` property contains a minimum and maximum value for a number input widget, and hence, a checking rule is created taking these values into account. The newly introduced utility class `ValidationUtils` creates the checking rules that need to be added by traversing the Structural UI Model tree and evaluating the `format` property of the screen's input widgets. These checking rules are then injected into the `<screenID>.js` file during its generation. The `<screenID>.js` file initializes the jQuery Validation plugin with the checking rules at run-time when the respective screen is loaded.

When submitting a form, the jQuery Validation plugin takes care of checking the relevant input elements against the defined rules. In case there have been errors while submitting the form, the corresponding error messages are generated. The newly created `validationUtil.js` module defines a function that receives the error messages from the jQuery Validation plugin and creates a container with a summary of all error messages, which is added to the top of the screen. Each error message is linked with its corresponding input field so that the input field is focused when the user clicks on it. `validationUtil.js` also places error messages directly next to their input fields and connects both of them semantically via the ARIA attribute `aria-labelledby`.

Because of the way UCP was built, there is only one global form on the Web page, but depending on which button the user clicks, only a part of the form should be checked for errors (see, e.g., the Shopping application, where only the elements within the panel the button belongs to should be checked). To do this, the relevant rules for the respective input elements are filtered before each form submission. `ValidationUtils` is used to traverse the Structural UI Model's component tree and collect all input widgets belonging to the same Communicative Act as the submit button that was activated. Those are the input widgets that need to be checked for form errors when the submit button is activated. In `<screenID>.js`, an event listener is set on each submit button, which performs the filtering before the actual form submission. This is done by temporarily removing the `required` attribute from the input widgets that should not be checked. After the jQuery Validation plugin's form checking function was invoked, the `required` attribute is added again to the input widgets.

Widget Replacements

For widget replacement, all the alternatives are rendered into the HTML page at first. To do this, a new template for alternatives has been added to the generation process that wraps the respective alternative into a `<div>` container with a unique ID.

Alternatives which do not match the criteria are then hidden from the screen when the Web page is loaded. This is done by keeping a map of DOM elements of all available alternatives per Display Alternative. Which alternative belongs to which Display Alternative is determined by the new utility class `DisplayAlternativeUtils`. On page load, the map of DOM elements is initialized and the available conditions are evaluated using a newly created JavaScript function. If an alternative matches the alternative, it is shown on the screen, otherwise it is hidden. When certain events are triggered, a re-evaluation of the conditions is performed and depending on the result, the Display Alternative is updated with the matching alternative.

The first condition that needs to match is whether the number of options (in a list widget) is greater than the value of the *"Rows Greater Than"* property of the Alternative widget in the Structural UI Model. This condition has a higher priority than the second condition, i.e., when the number of options is smaller than or equal to the property's value, the alternative will not be shown on the screen, no matter whether the second condition matches. The first condition is evaluated once when the page loads, since the number of options cannot change dynamically.

The second condition evaluates to true if the current screen width is within the range defined by the *"Minimum Screen Width"* and *"Maximum Screen Width"* properties of the Alternative widget. A media query string is built for each alternative based on these properties using `DisplayAlternativeUtils`. Each media query string is then injected into the JavaScript function responsible for evaluating the condition of the respective alternative. The media queries are evaluated once at page load and every time the browser window is resized (using an event listener on the `window.matchMedia` function which is triggered every time the browser window size changes).

Containers holding the alternatives' components are hidden via the CSS `display: none` property (as opposed to setting their `visibility`) to hide them from both the visible screen and screen readers.

CHAPTER 5

Evaluation and Results

The implementation of accessibility measures in UCP was evaluated by generating the Flight Booking Round Trip application in various variants (to cover the different types of widget replacements) and evaluating those variants against the WCAG 2.1. Section 5.1 describes the evaluation process. The results are given in Section 5.2.

5.1 Evaluation

Like in the case study in Chapter 3, the WCAG-EM was employed. The target guidelines were WCAG 2.1 up to Level AAA. For the evaluation, four variants of the Flight Booking Round Trip application have been generated using the adapted UCP framework that included the accessibility improvements discussed in Section 4.4:

1. A variant containing the “Tabs to Accordion” widget replacement, wrapping all the panels into their own tab. The tab view is replaced by an accordion view if the screen width is smaller than 620 pixels.
2. A variant containing the typical table form for displaying the departure and destination airports as well as the available flights.
3. A variant displaying the available flights, departure and destination airports in stacked tables.
4. A variant showing combo boxes for available flights, departure and destination airports.

Similarly to the case study, a combination of automatic evaluation, manual inspection, and a screening technique (i.e., using screen readers and a keyboard or touchscreen, respectively, while turning off the monitor) was used.

The evaluation was carried out on a desktop computer (running Ubuntu Linux 18.04 and Windows 10, respectively), a tablet (running iOS 12.1), and a smartphone (running Android 8.1). A mouse, a keyboard and three screen readers (JAWS 17, Voice Over and Google TalkBack) were used as input devices. The browsers being used were Chrome 65, Firefox 63, Internet Explorer 11, Edge 44, and Safari 12 (on iOS).

WAVE, SortSite Trial, Total Validator, and TAW were used as automatic accessibility evaluation tools. These tools are currently only checking for compliance with WCAG 2.0, i.e., the additional guidelines from WCAG 2.1 could only be evaluated manually. Additionally, the HTML and CSS validators have been applied.

The guidelines and documents being used as a reference for the accessibility evaluation were:

- WCAG 2.1 Guidelines (W3C Recommendation as of June 2018)
- How to Meet WCAG 2.1
- Understanding WCAG 2.1
- Techniques for WCAG 2.1
- WCAG2ICT
- Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile

Notice that the last two documents only take WCAG 2.0 into account.

5.2 Results

In the following, the results regarding each of the four WCAG 2.1 sections are discussed in detail. A summary including a table showing which success criteria were met can be found at the end of this section.

5.2.1 Perceivable

A text alternative is provided for all non-text content (e.g., names and labels for input fields, description text for the date picker and link icons accessible by screen readers).

Since time-based media is currently not supported in UCP, the guideline 1.2 Time-based Media does not apply.

Information, structure, and relationships can be perceived both visually and via screen readers. For example, panel headlines have a white text and blue background to distinguish them from normal text, while they are marked up using a HTML headline tag so that the screen reader interprets them as headlines.

The sequence of content is determined by the model designer. If he or she models the flow of the application in a meaningful way, then also the application content's are generated to be presented in a meaningful sequence.

Information is not limited to a certain sensory characteristic. For example, form errors are not indicated solely by an icon or a red color, they are also using markup like `aria-invalid` or `aria-describedby`. The date picker icon is placed in a button that contains a hidden text which is accessible to screen readers that says "Show calendar for [label name]".

Since the adapted version of UCP generates responsive applications, the content is not restricted to a certain display orientation. Component sizes and placements are adapted based on the available screen width.

The purpose of input fields is described by their label and input hints (if available). The purpose of other GUI components, icons or regions can also be determined by screen readers due to their markup.

No additional content is shown only when hovering or focusing an element (except text within the `title` attribute, which adheres to the guidelines).

Color contrast is high enough for level AA, requiring a contrast ratio of 4.5:1, and level AAA, requiring a contrast ratio of 7:1. Adjacent colors of non-text elements in the GUI (like input fields) have a contrast ratio of more than 3:1.

Text can be resized using browser settings as needed. Additionally, the whole GUI can be zoomed, while the content will adapt and reflow to fit into the screen without adding a horizontal scroll axis or overlapping contents. Adaptations of the styles can be made by users with custom stylesheets, since no styling is enforced by the generated application (e.g., using `!important`). Sensible defaults have been implemented (e.g., regarding line spacing, colors, and so on).

Since there are no audio controls or background audio, the guidelines "1.4.2 Audio Control" and "1.4.7 Low or No Background Audio" do not apply.

5.2.2 Operable

These guidelines relate to the operability of GUI components.

All GUI components of the adapted UCP framework can be operated via keyboard. The keyboard focus is never trapped, i.e., the focus can be removed from an element using the tab key. Keyboard shortcuts are used for some components like the date picker, where the user can conveniently switch, e.g., between months or years, using a certain key. However, these shortcuts are active only when focusing the respective GUI component, hence the relevant success criterion is met.

Since there are no time limits for viewing the application's content (e.g., automatic updates of text) and no animations or flashes, the guidelines "2.2 Enough Time" and "2.3 Seizures and Physical Reactions" do not apply to UCP.

Page titles are given via the `<title>` tag in the header of the HTML document. They reflect the name of the application, which is sufficient for meeting the success criterion, although it could be more descriptive by describing the content of the current screen (like "Flight Booking Round Trip: Select a Flight"). This could not be achieved because it is not clear beforehand which content appears on which screen. There would have to be a way for the model designer to manually specify more descriptive page titles.

The purpose of a link (e.g., the links in the footer) is given by the text (telling the user what to expect) as well as an icon and a hidden text (which is only accessible to screen readers) telling a person that it is a link referencing to an external source.

Section headings are implemented by the headings of the panels. There are various headings and labels to describe the content. Of course, the model designer needs to make sure that these headings and labels are descriptive by providing appropriate texts.

The focus is always visible. Regarding the focus order, UCP provides sensible defaults by not interfering with the natural focus order. However, the Flight Booking Round Trip Accessibility example application in Chapter 3 shows that the model designer could interfere with a sensible focus order by using custom rules to modify the order of content.

Hints about the user's location (like breadcrumbs, highlighting of the current page in the navigation) within an application could not be provided, since the navigation could not be implemented (see Section 4.2.8). The same applies to the success criterion requiring multiple ways to navigate an application and bypassing blocks of repeated content on each Web page.

Regarding input modalities:

- There are no GUI components that require multipoint or path-based gestures.
- There are no down events for activating a functionality, e.g., if the user has accidentally pressed a button, then moving out of the button area while having the button still pressed does not perform the button action.
- Programmatically accessible names match or start with the visible label of a GUI control.
- Functionality is not operated via device motion (e.g., shaking, tilting).
- The target size of 44x44 pixels was not reached. This is required by a level AAA success criterion. While the minimum target width of 44 pixels is met for all GUI controls (buttons, input fields, links,...), a minimum height of 44 pixels was not even met in the proof-of-concept application, where the larger target areas (e.g., the radio buttons in the table had a target height of 36 pixels) led to a lot more scrolling which was criticized by the users in the user study.
- The application does not restrict the use of input modalities (e.g., exclude keyboards or screen readers from accessing content or GUI components).

5.2.3 Understandable

With these guidelines, it should be checked whether the GUI is understandable for the users.

The first guideline deals with readability. The first success criterion, requiring the Web page to specify a default language that can be programmatically determined, is met by setting the language attribute of the HTML tag. It was defined that UCP currently supports only English, but it could be easily extended to support any other language by extending the model with a field asking the model designer for the language of the application.

The other success criteria of this guideline cannot be met solely by making the source code generated via UCP accessible. They require that the language must also be specified for parts of the GUI, which are not in the same language as the rest of the Web page. Additionally, unusual words (like idioms or jargon), abbreviations, and pronunciation (e.g., when the meaning of a word depends on its pronunciation) have to be clarified. Further, when a text is hard to understand for someone having just a lower secondary education level, an alternative version of the text that is easier to understand, has to be available. Since these success criteria strongly depend on the content and markup of the text that is provided by the model designer, he or she would need to make sure that the texts are readable.

The second guideline focuses on the predictability of the GUI. Its level A success criteria require that the context is not changed when a component receives focus or during user input. For example, opening a help dialog and focusing as soon as user focuses an input field, or showing a set of further input fields when a certain user input was given, would fail these success criteria. UCP meets these criteria, since neither focusing an input field nor providing input changes the context. Context-sensitive help messages (i.e., the input hints) are non-disruptive, since they are read by the screen reader when an input field is focused without changing the focus from the input field to the hint (this is done via `aria-describedby`).

Another success criterion is dealing with consistent navigation. Since the navigation has only been addressed in the proof-of-concept application, but not in the implementation in UCP, this is still an open issue. For example, in the Booking Kärnten application, the three buttons used as a “navigation” sometimes change their order on consecutive pages. They even change their position, i.e., in case they do not fit on the screen horizontally, they are aligned vertically.

Consistent identification, also required by the guideline, could be achieved in the adapted version of UCP. Components with the same functionality are consistent in their appearance and usage (e.g., combo boxes are operated the same way every time, the icon button for the date picker is always the same, error messages are presented in the same way). The technique for implementing this guidelines, i.e., using consistent labels, names, and alternative texts for components with the same functionality, has been applied.

The last success criterion, requiring that context changes are only initiated by the user or can be turned off if they are initiated automatically, has already been fulfilled by the original version of UCP and the adapted version still adheres to it.

The third guideline requires the GUI to provide help in order to avoid errors or correct mistakes. Form errors are detected on form submission and described textually as well as marked up correctly (using `aria-invalid`). Several advisory techniques (providing a summary of all errors at the top of the page, enabling the user to jump from the summary directly to an error, and so on) have been applied. Descriptive labels (using the `<label>` tag or `aria-labelledby`) and further input hints (using `aria-describedby`), and grouping of related input elements using `<fieldset>` were implemented. When a form error is detected, a suggestion for correction is displayed to the user (e.g., if no option was selected for a required radio button group, the error message suggests “Please choose an option”).

With regard to error prevention, forms are checked for errors when clicking on the “Submit” button, and when errors have occurred, the form is not submitted but instead, error messages are presented and the user can correct them. This basically meets the success criterion for error prevention, but only if all the relevant fields are marked as required (by the model developer). For example, in the Flight Booking Round Trip application, one could submit the form even though he did not fill in personal or credit card data. Hence, the model designer needs to make sure (and be aware) that the forms are modeled correctly. The same applies to providing context-sensitive help. In case some text input requires an expected data format, with the adapted Structural UI, a hint can be displayed that indicates this data format and provides an example. But again, the model designer needs to provide such examples, or else they will not be shown.

5.2.4 Robust

The goal of this WCAG section with regard to Web applications is that they support a wide variety of (current and future) browsers and assistive technologies.

To ensure this, the markup has to be correct according to the specification. For example, HTML elements have to be correctly opened and closed, no duplicate IDs should be used and elements should be nested according to the specification. In the original version of UCP, in some cases, identical IDs were assigned to multiple elements, and HTML attributes were not always wrapped between double quotes. These issues have been resolved in the new version of UCP developed for this thesis. Running the HTML and CSS validators on the variations of the Flight Booking Round Trip application showed no such errors.

Another aspect of robustness is that names, roles, states, properties, and values of GUI components can be programmatically determined, and that assistive technologies are notified when these elements change. This success criterion is now fulfilled for the adapted UCP framework. For example, the revised tab component now exposes the correct roles and states to determine its status: the roles `tablist`, `tab`, and `tabpanel` are assigned to the correct HTML elements, and state changes are indicated via the ARIA attributes `aria-selected` and `aria-hidden`. In cases where there are no visible labels (like for choosing the departure and return dates in the Flight Booking Round Trip application, since the purpose of the input field is already given by the panel's title), the `aria-label` attribute is used to provide an implicit label that is accessible by assistive technologies.

The last aspect is that assistive technology needs to be notified about changing status messages without receiving focus. This includes, for example, the number of results available for a given input in the combo box. The focus remains in the input field, where the user types in his or her desired option, while the screen reader reads the number of options found in the combo box matching the user input. This was achieved in the adapted UCP framework by using the `aria-live` attribute on an element containing, e.g., the text `"3 results found"`. Whenever the user types in a character, this element's text is updated with the results found and read to the user by the screen reader.

5.2.5 Summary

Overall, many of the issues (67 %) found in the original version of UCP could be resolved. These issues were mainly related to the "Perceivable" and "Robust" guidelines, for which all issues have been resolved. The most significant improvement could be made for level A accessibility issues, where 94 % issues were eliminated.

However, the adapted version of UCP failed to meet 10 success criteria of the WCAG 2.1, which can be seen in Figure 5.1. All of them relate to the "Operable" (six issues) and "Understandable" (four issues) guidelines. One level A issue, three level AA issues, and six level AAA issues could not be met.

The reason that there are four level AAA issues for the "Understandable" guideline as compared to only one level AAA issue discovered in the case study in Chapter 3, is that in the case study, no examples existed that required explanation of unusual words,

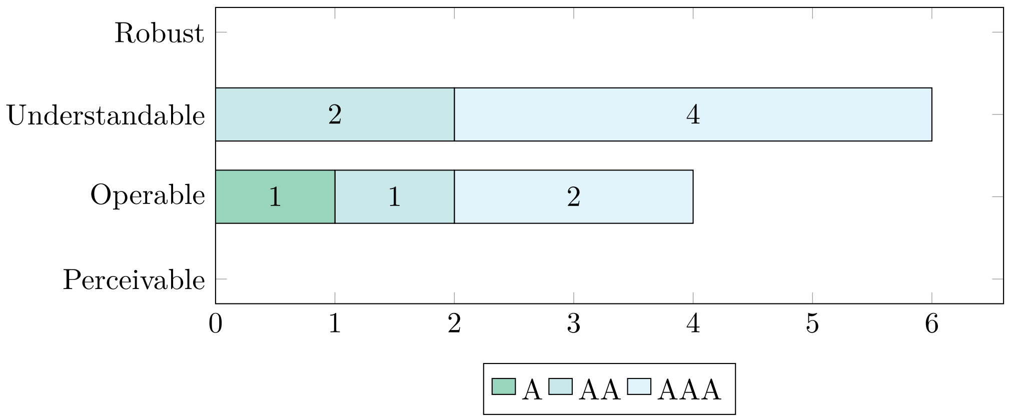


Figure 5.1: Distribution of discovered accessibility issues in the adapted UCP version among the WCAG 2.1 sections

abbreviations, or that required a reading level above secondary education level), so these issues have not been discovered in the case study, although they existed.

The success criteria that could not be met by the adapted version of UCP were:

- 2.4.1 Bypass Blocks (A)
- 2.4.5 Multiple Ways (AA)
- 2.4.8 Location (AAA)
- 2.5.5 Target Size (AAA)
- 3.1.2 Language of Parts (AA)
- 3.1.3 Unusual Words (AAA)
- 3.1.4 Abbreviations (AAA)
- 3.1.5 Reading Level (AAA)
- 3.1.6 Pronunciation (AAA)
- 3.2.3 Consistent Navigation (AA)

Some success criteria can only be fulfilled if the model designer pays attention to them. If done so, UCP provides measures to adhere to the guideline. This applies to the following success criteria:

- 1.3.2 Meaningful Sequence (A)
- 2.4.3 Focus Order (A)
- 2.4.6 Headings and Labels (AA)
- 3.3.4 Error Prevention (Legal, Financial, Data) (AA)
- 3.3.5 Help (AAA)
- 3.3.6 Error Prevention (All) (AAA)

The following success criteria do not apply to the adapted version of UCP, since the features being addressed are currently not being generated by the framework:

- 1.2 Time-based Media
- 1.4.2 Audio Control
- 1.4.7 Low or No Background Audio
- 2.2 Enough Time
- 2.3 Seizures and Physical Reactions

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

The adapted version of UCP shows that several accessibility issues can be addressed by frameworks that automatically generate GUIs at design-time. If there are no repeated blocks of content in the application to generate (e.g., navigation or header bars), and the model designer structures the model in a way that leads to a meaningful sequence of content, level A of the WCAG can be reached by the resulting GUI. Moreover, process-oriented applications that use only one language, with descriptive headings and labels as well as input hints, both defined by the model designer, can even reach level AA of the WCAG. The Flight Booking Round Trip application, which was taken as the primary example in this thesis, reaches level A as is, and could reach level AA if the personal and credit card details were defined as mandatory fields (which is possible in UCP but was not specified by the model designer).

From the results, it can be seen that design-time GUI generation frameworks can eliminate accessibility issues, especially when it comes to the "Perceivable" and "Robust" guidelines and level A criteria of the WCAG. These guidelines mainly address accessibility features in the source code, which can easily be controlled by building the framework in a way that generates the source code correctly. The big advantage of GUI generation frameworks over manual development of GUIs is that the resulting (accessible) GUI source code is always reproducible. When generating the GUI automatically, one will always receive a result of the same quality with regard to the source code (i.e., all the accessibility features that the framework is capable of are available), and human error (e.g., forgetting to add a label to an input element) is avoided.

Another advantage of automatic GUI generation with regard to accessibility is that generated GUIs are consistent, i.e., that if a user knows how to operate one automatically generated GUI, he or she can also easily operate other GUIs that were generated by the same framework.

Device tailoring for automatic GUI generation makes sure that the number of GUI components on a screen and their placement is optimized to the available space. This by itself can reduce scrolling and hence lead to better accessibility. In this thesis, device tailoring was combined with responsive design, which resizes and reflows GUI components as needed to fit the available screen width. This enables the user to zoom while preserving the tailoring of the layout (i.e., elements will still fit into the available screen and scrolling in two direction is avoided). Creating responsive applications manually can be cumbersome and error-prone, since one has to think about how the components should reflow. Automatically generating responsive applications at design-time is a novel approach, which is efficient and consistent. On the other hand, it makes the automatic generation process of optimally tailored GUIs more difficult, since the ways in which GUI components can resize and reflow at run-time depending on the available space needs to be taken into account. For example, developers of frameworks like UCP need to think about the breakpoints for reflowing content from a multi-column layout to a single-column layout or when it makes sense to hide certain (less relevant) content on smaller screens to save screen space.

The same applies to widget replacement. In this thesis, various widget replacements have been proposed in order to optimize the presentation of contents depending on the available space or amount of information. This can improve accessibility by saving screen space, avoid scrolling in two directions and making the components easier to understand. Again, this adds more complexity to the generation process, since the various conditions for widget replacements need to be defined carefully and also the model becomes more complex.

The approach of using discourse models (or other high-level models in general) enables the model designer to enrich the source code with additional information like texts for screen readers, without having to know how they need to be marked up to be accessible. Automatic GUI generation turns the focus on business problems by empowering domain experts, since modeling can be done without requiring knowledge in programming. However, like accessibility evaluation, the implementation of accessibility measures cannot be fully automated. The model designer also needs to be educated about accessibility concerns, since some accessibility measures are directly reflected in the models. The results of this thesis show that some success criteria of the WCAG, ranging over all levels, can only be fulfilled if the model designer pays attention to them and provides sensible information in the models (e.g., descriptive headings and labels, input hints, or logical structuring of the content flow). Hence the model designer could render the application inaccessible by unwittingly missing to provide necessary information. This also applies to custom rules and custom widgets, on which the GUI generation framework has no (or only limited) influence.

The thesis also showed that software aging is important to consider when dealing with accessibility for automated GUI generation framework. The technologies and mechanisms for generating the FUI have not been updated in UCP for several years. HTML table layouts were still a widely-used layouting technique at the time they were introduced to

UCP, but are now considered out-dated and even harmful when it comes to accessibility. Moreover, in the course of writing this thesis, the WCAG guidelines were in the process of being updated from version 2.0 to version 2.1, adding several new success criteria that needed to be taken into account. In a few years, W3C project “Silver” will completely revise the WCAG, leading to substantial changes in the guidelines that need to be addressed. Hence, automated GUI generation frameworks need to keep up with changes of the guidelines and technologies and should be updated accordingly in a timely manner.

One needs to consider that GUI generation frameworks are only one part of a broader eco-system. Model designers (or content creators) are another part, and user agents (like browsers or screen readers), for which their own set of guidelines (i.e., the UAAG) exist, are yet another. All these three entities need to play together to provide an accessible experience to the user. Moreover, the quality of the accessibility guidelines also plays a major role when creating accessible applications. Some argue that the WCAG are complex and hard to interpret, and do not provide enough guidance for solving accessibility issues (e.g., [BMSF16], [OTD16]). Due to lack of other widely-adopted accessibility guidelines and the manifestation of the WCAG in various laws, one does not really have an alternative option.

6.2 Future Work

In January 2019, a second user study similar to the one included in this thesis is being conducted comparing two versions of the Flight Booking Round Trip application generated with the adapted version of UCP and viewed on a smartphone. The first one contains typical table views for showing radio button options, while the other contains dropdowns for showing the options. This study shall reveal which version is preferred by users and whether it makes sense to perform the “Radio Button to Dropdown” widget replacement on smartphones.

Studies involving users with certain disabilities could help identifying further accessibility improvements that could be implemented within UCP. In a related study [RTKP18] it was claimed that the “Radio Buttons to Dropdown” widget replacement may improve low-vision accessibility. This could be examined by conducting a user study involving participants with various kinds of disabilities of sight (like glaucoma, cataract, or macula degeneration).

With regard to UCP, the process of automatically generating the adapted Structural UI Model needed to generate accessible GUIs is still an open issue. In this thesis, the Structural UI Model that is currently generated by UCP was manually modified in order to automatically generate the accessible source code. These manual modifications include display alternatives used for widget replacements, table headers, input hints and the connection between labels and their respective input fields. Measures of how to add this information to the Discourse Model or to infer it from the available data (e.g., creating the display alternatives based on how many options there are in the data model) need to be investigated and implemented in UCP.

A question that still remains is how many media breakpoints for responsive design are needed to provide the best accessibility. In this thesis, the assumption was made that two breakpoints are sufficient, one switching between a view tailored for desktop computers and a view tailored for tablets, and the other switching between views tailored for tablets and smartphones, respectively. This presumption could be accepted or refuted in future work.

Moreover, more detailed information for responsive design could be supplied by the Structural UI Model (e.g. whether a three-column-layout on a desktop screen should reflow to a two-column-layout on a tablet screen by specifying the class names `col-lg-4` and `col-md-6`, respectively, using layout hints).

In case content includes parts in a different language, unusual words, abbreviations, or pronunciations, UCP should either provide means for the model designer to indicate them or leverage techniques and methods like Natural Language Processing (NLP) to automatically detect these content parts and simplify the text content. There has been research suggesting how to detect such parts and simplify text using NLP [HWLJMR11][MMSBR15], but integrating it into UCP is still an open issue.

Furthermore, the problem of providing correct markup and multiple ways for navigation that is consistent and can be skipped is another open issue in UCP. A solution of how such a navigation could look like was provided by the proof-of-concept application in this thesis, but implementing it in UCP would require substantial changes of its runtime framework.

In case UCP includes generating players for audio or video content, or supports dynamic or time-based content (like animations) in the future, the WCAG guidelines that currently do not apply to applications generated by UCP need to be addressed. For example, UCP would need to provide a way for the model designer to specify audio transcriptions and the generated source code would have to provide a way to turn off animations.

In recent years, related work tends to focus on context-aware adaptive GUIs that change their appearance in response to certain user behaviors and preferences (e.g., [MPAA16], [YSSE17], [HUHMB$^+$18]). This requires adaptations of the GUI at run-time. With responsive design and widget replacement, this thesis already introduced some run-time adaptations to UCP. Future work could augment the GUI that UCP generates at design-time even further by, for example, adding user tracking techniques to detect which disability one user has to decide from which widget replacement the user would benefit most (as suggested in [RTKP18]). In this way, a generally accessible baseline GUI could be generated at design-time, which is then personalized to a certain user's needs to improve accessibility for the respective user even further. Some accessibility measures targeted for one disability might not be optimal for a user having a different disability. With a combined approach of design-time generation and run-time adaptation and personalization, the GUI could provide exactly those accessibility measures that each user needs.

List of Figures

Acronyms

ANM Action-Notification Model. 36

API Application Programming Interface. 26, 27

AT-SPI Assistive Technology Service Provider Interface. 27

ATAG Authoring Tool Accessibility Guidelines. 16

AUI Abstract User Interface. 34, 37, 39, 40

AVB-IT General conditions of contract for IT services and software of the Austrian republic. 13

CIM Computation Independent Model. 34, 41

CRF CAMELEON Reference Framework. 34–37, 39, 42, 123

CSS Cascading Style Sheets. 18–21, 23, 24, 31, 46, 51, 54, 58, 60, 61, 63, 66, 68, 75, 81, 104, 106–108, 110, 112, 116

CUI Concrete User Interface. 34, 37, 39, 42, 70

DoD Domain-of-Discourse. 36, 51

DOM Document Object Model. 22, 27, 107, 110

DSL Domain Specific Language. 36

EARL Evaluation and Report Language. 30, 32

EU European Union. 5, 13, 14, 30

FUI Final User Interface. 34, 42, 43, 97, 100, 120

GUI Graphical User Interface. 1–3, 5, 9–12, 19, 21, 22, 24, 26, 27, 32–35, 37–45, 47–51, 58, 69–71, 73–77, 90, 92, 95–97, 100, 101, 104, 106, 113–116, 119–122

HTML Hypertext Markup Language. 8, 16, 17, 19–21, 23, 24, 31, 38, 40, 41, 54, 58, 60, 61, 63, 65, 85, 104, 106–110, 112, 114–116, 120

ICT Information and Communications Technology. 5, 6, 8, 9, 11–18, 32, 53, 85

IDE Integrated Development Environment. 35, 43

ISO International Organization for Standardization. 15, 17

MDA Model Driven Architecture. 33, 34, 41

MSAA Microsoft Active Accessibility. 27

MVC Model-View-Controller. 35

NLP Natural Language Processing. 122

OMG Object Management Group. 33

PIM Platform Independent Model. 34

PSM Platform Specific Model. 34

RCP Rich Client Platform. 35

RIA Rich Internet Application. 39

RIAs Rich Internet Applications. 16

RST Rhetorical Structure Theory. 37

RWD Responsive Web Design. 18, 19, 74, 92

SEO Search Engine Optimization. 12

UAAG User Agent Accessibility Guidelines. 16, 121

UCP Unified Communication Platform. 2, 3, 5, 21, 22, 34–36, 38, 45–49, 53–59, 62, 63, 69–71, 73, 74, 76, 77, 81, 82, 85, 91, 92, 96, 97, 99–101, 104, 106–109, 111–125

UIA UI Automation. 27

UIDL User Interface Description Language. 40, 42

UML Unified Modeling Language. 36, 37

UN United Nations. 5, 13, 14

UWEM Unified Web Evaluation Methodology. 30

W3C World Wide Web Consortium. 15–17, 19, 25, 57, 60, 61, 67, 68, 80, 112, 121

WAI Web Accessibility Initiative. 15, 16, 28

WCAG Web Content Accessibility Guidelines. 6, 11, 14–18, 22, 25, 27–30, 38, 40, 41, 49, 53–58, 60–62, 65–69, 111, 112, 116, 117, 119–123, 125

WCAG-EM Website Accessibility Conformance Evaluation Methodology. 28–30, 53–55, 111, 123

WHO World Health Organization. 7

WIMP Window, Icon, Menu, Pointer. 35

Bibliography

[AAC+11] Julio Abascal, Amaia Aizpurua, Idoia Cearreta, Borja Gamecho, Nestor Garay-Vitoria, and Raúl Miñón. Automatically generating tailored accessible user interfaces for ubiquitous services. In *The proceedings of the 13th international ACM SIGACCESS conference on Computers and accessibility*, pages 187–194. ACM, 2011.

[AaIV08] Silvia Abrahão, Emilio Iborra, and Jean Vanderdonckt. Usability evaluation of user interfaces generated with a model-driven architecture tool. In EffieLai-Chong Law, EbbaThora Hvannberg, and Gilbert Cockton, editors, *Maturing Usability*, Human-Computer Interaction Series, pages 3–32. Springer London, 2008. URL: `http://dx.doi.org/10.1007/978-1-84628-941-5_1`, `doi:10.1007/978-1-84628-941-5\_1`.

[ASF10] The Apache Software Foundation. *The Apache Velocity Project*, 2010. URL: `https://velocity.apache.org/` [cited 2016-10-31].

[AVCF+10] Nathalie Aquino, Jean Vanderdonckt, Nelly Condori-Fernández, Óscar Dieste, and Óscar Pastor. Usability evaluation of multi-device/platform user interfaces generated by model-driven engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 30:1–30:10, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1852786.1852826`, `doi:10.1145/1852786.1852826`.

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

[BMA13] Begleitgruppe zum nationalen Aktionsplan Behinderung 2012-2020. 2013. URL: `https://www.sozialministerium.at/cms/site/attachments/5/1/5/CH3434/CMS1450699435356/statistik_-_menschen_mit_behinderung_20131.pdf` [cited 2016-12-21].

[BMSF16] Aleksander Bai, Heidi Camilla Mork, Trenton Schulz, and Kristin Skeide Fuglerud. Evaluation of accessibility testing methods. which methods un-cover what type of problems? *Studies in health technology and informatics*, 229:506, 2016.

[Bra08] Giorgio Brajnik. Beyond conformance: the role of accessibility evaluation methods. In *Web Information Systems Engineering–WISE 2008 Workshops*, pages 63–80. Springer, 2008.

[BS15] P Benda and M Smejkalová. Web interface for education of mentally disabled persons for work in horticulture. *Agris on-line Papers in Economics and Informatics*, 7(1):13, 2015.

[BYH10] Giorgio Brajnik, Yeliz Yesilada, and Simon Harper. Testability and validity of wcag 2.0: the expertise effect. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*, pages 43–50. ACM, 2010.

[CCGV08] Ben Caldwell, Michael Cooper, Loretta Guarino Reid, and Gregg Vanderheiden. Web Content Accessibility Guidelines 2.0, 2008. URL: `https://www.w3.org/TR/UNDERSTANDING-WCAG20/appendixE.html#WCAG20` [cited 2016-12-17].

[CCT02] Gaëlle Calvary, Joëlle Coutaz, and David Thevenin. The CAMELEON Reference Framework. Technical report, 2002. URL: `http://giove.isti.cnr.it/projects/cameleon/pdf/CAMELEON%20D1.1RefFramework.pdf` [cited 2017-01-29].

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003. URL: `http://www.sciencedirect.com/science/article/pii/S0953543803000109`, doi:10.1016/S0953-5438(03)00010-9.

[CGV+18] Ben Caldwell, Loretta Guarino Reid, Gregg Vanderheiden, Wendy Chisholm, John Slatin, and Jason White. Web Content Accessibility Guidelines (WCAG) 2.1, 2018. URL: `https://www.w3.org/TR/WCAG21/` [cited 2018-07-01].

[CKAV13] Michael Cooper, Peter Korn, Snow-Weaver Andi, and Gregg Vanderheiden. Guidance on Applying WCAG 2.0 to Non-Web Information and Communications Technologies (WCAG2ICT), 2013. URL: `https://www.w3.org/TR/wcag2ict/` [cited 2017-01-06].

[CS04] Lawrence Chung and Sam Supakkul. Representing NFRs and FRs: A goal-oriented and use case driven approach. In *International Conference on Software Engineering Research and Applications*, pages 29–41. Springer, 2004.

[CS10] Adam Connors and Bryan Sullivan. Mobile Web Application Best Practices, 2010. URL: `https://www.w3.org/TR/mwabp/` [cited 2017-01-18].

[CVC08] Benoît Collignon, Jean Vanderdonckt, and Gaëlle Calvary. Model-driven engineering of multi-target plastic user interfaces. In *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems (ICAS 2008)*, pages 7–14, Washington, DC, USA, 2008. IEEE Computer Society. `doi:http://dx.doi.org/10.1109/ICAS.2008.37`.

[DJSV05] Nina Dahlmann, Sabina Jeschke, Ruedi Seiler, and Helmut Vieritz. Be-learning: Accessibility in virtual knowledge spaces for mathematics and natural sciences. 2005.

[DMV$^+$16] Carlos Duarte, Inês Matos, João Vicente, Ana Salvado, Carlos M Duarte, and Luís Carriço. Development technologies impact in web accessibility. In *Proceedings of the 13th Web for All Conference*, page 6. ACM, 2016.

[dOFP$^+$14] Roberto Cícero de Oliveira, André Pimenta Freire, Débora Maria Barroso Paiva, Maria Istela Cagnin, and Hana Rubinsztejn. A framework to facilitate the implementation of technical aspects of web accessibility. In *International Conference on Universal Access in Human-Computer Interaction*, pages 3–13. Springer, 2014.

[Ecl07] Eclipse Foundation. *Model to Text (M2T)*, 2007. URL: `https://eclipse.org/modeling/m2t/?project=jet` [cited 2016-10-31].

[EN315] EN 301 549 V1.1.2 Accessibility requirements suitable for public procurement of ICT products and services in Europe, 2015.

[FKH$^+$06] Jürgen Falb, Hermann Kaindl, Helmut Horacek, Cristian Bogdan, Roman Popp, and Edin Arnautovic. A discourse model for interaction design based on theories of human communication. In *Extended Abstracts on Human Factors in Computing Systems (CHI '06)*, pages 754–759. ACM Press: New York, NY, 2006. `doi:http://doi.acm.org/10.1145/1125451.1125602`.

[FPAP03] Joan Fons, Vicente Pelechano, Manoli Albert, and Oscar Pastor. Development of web applications from web enhanced conceptual schemas. In *International Conference on Conceptual Modeling*, pages 232–245. Springer, 2003.

[FPR$^+$07] Jürgen Falb, Roman Popp, Thomas Röck, Helmut Jelinek, Edin Arnautovic, and Hermann Kaindl. UI prototyping for multiple devices through specifying interaction design. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT 2007)*, pages 136–149, Rio de Janeiro, Brazil, September 2007. Springer.

[GGMM$^+$13] María González-García, Lourdes Moreno, Paloma Martínez, Raúl Miñon, and Julio Abascal. A model-based graphical editor to design accessible media players. *J. UCS*, 19(18):2656–2676, 2013.

[GGMM15] María González-García, Lourdes Moreno, and Paloma Martínez. A model-based tool to develop an accessible media player. In *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility*, pages 415–416. ACM, 2015.

[GK09] Jason Grieves and Masahiko Kaneko. *Engineering Software for Accessibility.* 2009.

[GKJ$^+$08] Peter Göhner, Simon Kunz, Sabina Jeschke, Helmut Vieritz, and Olivier Pfeiffer. Integrated Accessibility Models of User Interfaces for IT and Automation Systems. In *CAINE*, pages 280–285. Citeseer, 2008.

[GLW06] Krzysztof Z Gajos, Jing Jing Long, and Daniel S Weld. Automatically generating custom user interfaces for users with physical disabilities. In *Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 243–244. ACM, 2006.

[GMM15] María González, Lourdes Moreno, and Paloma Martínez. Approach design of an accessible media player. *Universal Access in the Information Society*, 14(1):45–55, 2015.

[GWW07] Krzysztof Z Gajos, Jacob O Wobbrock, and Daniel S Weld. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 231–240. ACM, 2007.

[HAS$^+$15] Elyse C. Hallett, Blake Arnsdorff, John Sweet, Zach Roberts, Wayne Dick, Tom Jewett, and Kim-Phuong L. Vu. The usability of magnification methods: A comparative study between screen magnifiers and responsive web design. In Sakae Yamamoto, editor, *Human Interface and the Management of Information. Information and Knowledge Design*, pages 181–189, Cham, 2015. Springer International Publishing.

[HUHMB$^+$18] Jamil Hussain, Anees Ul Hassan, Hafiz Syed Muhammad Bilal, Rahman Ali, Muhammad Afzal, Shujaat Hussain, Jaehun Bang, Oresti Banos, and Sungyoung Lee. Model-based adaptive user interface based on context and user experience evaluation. *Journal on Multimodal User Interfaces*, 12(1):1–16, Mar 2018. URL: `https://doi.org/10.1007/s12193-018-0258-2`, `doi:10.1007/s12193-018-0258-2`.

[HW16] Sabine Hennig and Wolfgang W Wasserburger. Design Patterns für barrierefreie Online-Karten. pages 308–317, 2016.

[HWLJMR11] Jeffery Higginbotham, Gregory W Lesher, Bryan J Moulton, and Brian Roark. The application of natural language processing to augmentative and alternative communication. *Assistive technology : the official journal of RESNA*, 24:14–24, 04 2011. `doi:10.1080/10400435.2011.648714`.

[ISOa] ISO 9241-11:1998 Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability.

[ISOb] ISO/IEC 40500:2012 Information technology – W3C Web Content Accessibility Guidelines (WCAG) 2.0.

[JPV09] Sabina Jeschke, Olivier Pfeiffer, and Helmut Vieritz. Using web accessibility patterns for web application development. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 129–135. ACM, 2009.

[JV17] J. Bern Jordan and Gregg C. Vanderheiden. Towards accessible automatically generated interfaces part 1: An input model that bridges the needs of users and product functionality. In Jia Zhou and Gavriel Salvendy, editors, *Human Aspects of IT for the Aged Population. Aging, Design and User Experience*, pages 129–146, Cham, 2017. Springer International Publishing.

[Ken02] Stuart Kent. Model driven engineering. In *International Conference on Integrated Formal Methods*, pages 286–298. Springer, 2002.

[KPR12] Hermann Kaindl, Roman Popp, and David Raneburger. Automated generation of user interfaces: Based on use case or interaction design specifications? In Slimane Hammoudi, Marten van Sinderen, and José Cordeiro, editors, *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT'12)*, pages 303–308. SciTePress, July 2012.

[KRF$^+$09] Sevan Kavaldjian, David Raneburger, Jürgen Falb, Hermann Kaindl, and Dominik Ertl. Semi-automatic user interface generation considering pointing granularity. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009.

[LFG90] Paul Luff, David Frohlich, and Nigel Gilbert. *Computers and Conversation.* Academic Press, London, UK, January 1990.

[LLTPT$^+$11] Marino Linaje, Adolfo Lozano-Tello, Miguel A Perez-Toledano, Juan Carlos Preciado, Roberto Rodriguez-Echeverria, and Fernando Sanchez-Figueroa. Providing ria user interfaces with accessibility properties. *Journal of Symbolic Computation*, 46(2):207–217, 2011.

[LPSF07] Marino Linaje, Juan Carlos Preciado, and Fernando Sánchez-Figueroa. Engineering rich internet application user interfaces over legacy web models. *IEEE internet computing*, 11(6), 2007.

[Mar10] Ethan Marcotte. Responsive web design, 2010. *URL http://alistapart.com/article/responsive-web-design*, 2010.

[MDA14] MDA guide version 2.0. Technical report, Object Management Group (OMG), 2014. URL: `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01` [cited 2017-01-29].

[MFT05] Jennifer Mankoff, Holly Fait, and Tu Tran. Is your web page accessible?: a comparative study of methods for assessing web page accessibility for the blind. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 41–50. ACM, 2005.

[MMA13] Raúl Miñón, Lourdes Moreno, and Julio Abascal. A graphical tool to create user interface models for ubiquitous interaction satisfying accessibility requirements. *Universal access in the information society*, 12(4):427–439, 2013.

[MMMA14] Raúl Miñón, Lourdes Moreno, Paloma Martínez, and Julio Abascal. An approach to the integration of accessibility requirements into a user interface development method. *Science of Computer Programming*, 86:58–73, 2014.

[MMR08a] Lourdes Moreno, Paloma Martínez, and Belén Ruiz. A MDD approach for modelling web accessibility. In *ICWE 2008 Workshops*, page 7, 2008.

[MMR08b] Lourdes Moreno, Paloma Martínez, and Belén Ruiz. Guiding accessibility issues in the design of websites. In *Proceedings of the 26th annual ACM international conference on Design of communication*, pages 65–72. ACM, 2008.

[MMSBR15] Lourdes Moreno, Paloma Martínez, Isabel Segura-Bedmar, and Ricardo Revert. Exploring language technologies to provide support to wcag 2.0 and e2r guidelines. In *Proceedings of the XVI International Conference on Human Computer Interaction*, page 57. ACM, 2015.

[MPAA16] Raúl Miñón, Fabio Paternò, Myriam Arrue, and Julio Abascal. Integrating adaptation rules for people with special needs in model-based UI development process. *Universal Access in the Information Society*, 15(1):153–168, 2016.

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM. URL: http://doi.acm.org/10.1145/142750.142789, doi:10.1145/142750.142789.

[MRCG10] Adriana Martín, Gustavo Rossi, Alejandra Cechich, and Silvia Gordillo. Engineering accessible Web applications. An aspect-oriented approach. *World Wide Web*, 13(4):419–440, 2010.

[MSMG12] Adriana Martin, Viviana Saldaño, Gabriela Miranda, and Gabriela Gaetan. AO-WAD: A Generalized Approach for Accessible Design within the Development of Web-based Systems. In *Proceedings of The 7th International Conference on Software Engineering Advances, ICSEA*, pages 581–587, 2012.

[MT88] W. C. Mann and S.A. Thompson. Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[OAS07] Zeljko Obrenovic, Julio Abascal, and Dusan Starcevic. Universal accessibility as a multimodal design issue. *Communications of the ACM*, 50(5):83–88, 2007.

[OTD16] Anyela Orozco, Valentina Tabares, and Néstor Duque. Methodology for heuristic evaluation of web accessibility oriented to types of disabilities. In *International Conference on Universal Access in Human-Computer Interaction*, pages 91–97. Springer, 2016.

[Pet07] Roland Petrasch. Model Based User Interface Design: Model Driven Architecture. 2007.

[PHJS12] Matthias Peissner, Dagmar Häbe, Doris Janssen, and Thomas Sellner. Myui: generating accessible user interfaces from multimodal design patterns. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 81–90. ACM, 2012.

[PKR12] TU Wien. *Unified Communication Platform*, 2012. URL: https://ucp.ict.tuwien.ac.at/ [cited 2016-10-13].

[Pop12] Roman Popp. A unified solution for service-oriented architecture and user interface generation through discourse-based communication models. Doctoral dissertation, Vienna University of Technology, Vienna, Austria, 2012.

[PRK13] Roman Popp, David Raneburger, and Hermann Kaindl. Tool support for automated multi-device GUI generation from discourse-based communication models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, New York, NY, USA, 2013. ACM.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16:19:1–19:30, November 2009. URL: http://doi.acm.org/10.1145/1614390.1614394, doi:http://doi.acm.org/10.1145/1614390.1614394.

[PSW15] Kim Patch, Jeanne Spellman, and Kathy Wahlbin. Mobile Accessibility: How WCAG 2.0 and Other W3C/WAI Guidelines Apply to Mobile, 2015. URL: https://www.w3.org/TR/mobile-accessibility-mapping/ [cited 2017-01-18].

[RKP15a] David Raneburger, Hermann Kaindl, and Roman Popp. Model transformation rules for customization of multi-device graphical user interfaces. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, pages 100–109, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2774225.2774839, doi:10.1145/2774225.2774839.

[RKP15b] David Raneburger, Hermann Kaindl, and Roman Popp. Strategies for automated GUI tailoring for multiple devices. In *Proceedings of the 48th Annual Hawaii International Conference on System Sciences (HICSS-48)*, pages 507–516, Piscataway, NJ, USA, 2015. IEEE Computer Society Press.

[RM08] Jo Rabin and Charles McCathieNevile. Mobile Web Best Practices 1.0 – Basic Guidelines, 2008. URL: https://www.w3.org/TR/mobile-bp/ [cited 2017-01-18].

[RPF05] Gonzalo Rojas, Vicente Pelechano, and Joan Fons. A model-driven approach to include adaptive navigational techniques in web applications. In *International Workshop on Web Oriented Software Technology (IWWOST)*, pages 13–24, 2005.

[RPK$^+$11] David Raneburger, Roman Popp, Sevan Kavaldjian, Hermann Kaindl, and Jürgen Falb. Optimized GUI generation for small screens. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 107–122. Springer Berlin / Heidelberg, 2011.

[RPK16] Thomas Rathfux, Roman Popp, and Hermann Kaindl. Adding custom widgets to model-driven GUI generation. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, pages 16–26, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2933242.2933251, doi:10.1145/2933242.2933251.

[RPV12] David Raneburger, Roman Popp, and Jean Vanderdonckt. An automated layout approach for model-driven WIMP-UI generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 91–100, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2305484.2305501`, `doi:10.1145/2305484.2305501`.

[RTKP18] Thomas Rathfux, Jasmin Thöner, Hermann Kaindl, and Roman Popp. Combining design-time generation of web-pages with responsive design for improving low-vision accessibility. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 10. ACM, 2018.

[RV16] Predrag Radic and Edith Vosta. Berücksichtigung der Barrierefreiheit bei Auftragsvergaben - Planen und Umsetzen barrierefreier IKT-Lösungen, 2016. URL: `https://www.ag.bka.gv.at/at.gv.bka.wiki-bka/img{_}auth.php/6/6b/VM2016{_}Barrierefreiheit{_}bei{_}Auftragsverfahren.pdf` [cited 2016-12-17].

[Sch06] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006. `doi:10.1109/MC.2006.58`.

[Sea69] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, England, 1969.

[SF$^+$07] F Sánchez-Figueroa et al. Saw, a set of integrated tools for making the web accessible to visually impaired use. *UPGRADE*, 8, 2007.

[Tru06] Frank Truyen. The Fast Guide to Model Driven Architecture - The basics of Model Driven Architecture, January 2006. URL: `http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf`.

[VAZ14] Eric Velleman and Shadi Abou-Zahra. Website Accessibility Conformance Evaluation Methodology (WCAG-EM) 1.0, 2014. URL: `https://www.w3.org/TR/WCAG-EM/` [cited 2017-01-26].

[VBC13] Markel Vigo, Justin Brown, and Vivienne Conway. Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility*, page 1. ACM, 2013.

[VJP11] Helmut Vieritz, Sabina Jeschke, and Olivier Pfeiffer. Benutzungsorientierte Entwicklung barrierefreier Benutzungsschnittstellen. In *Automation, Communication and Cybernetics in Science and Engineering 2009/2010*, pages 569–578. Springer, 2011.

[VPJ07] Helmut Vieritz, Olivier Pfeiffer, and Sabina Jeschke. Belearning: Designing accessible elearning applications. In *Frontiers In Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE'07. 37th Annual*, pages S3D–1. IEEE, 2007.

[VSDS00] Patrícia Vilain, Daniel Schwabe, and Clarisse Sieckenius De Souza. A diagrammatic tool for representing user interaction in uml. In *International Conference on the Unified Modeling Language*, pages 133–147. Springer, 2000.

[W3Ca] World Wide Web Consortium (W3C). *About W3C*. URL: `https://www.w3.org/Consortium/` [cited 2017-01-04].

[W3Cb] World Wide Web Consortium (W3C). *Accessibility, Usability, and Inclusion: Related Aspects of a Web for All*. URL: `https://www.w3.org/WAI/intro/usable` [cited 2016-12-17].

[W3Cc] World Wide Web Consortium (W3C). *HTML5 - A vocabulary and associated APIs for HTML and XHTML*. URL: `https://www.w3.org/TR/html5/` [cited 2016-10-31].

[W3Cd] World Wide Web Consortium (W3C). *Media Queries*. URL: `https://www.w3.org/TR/css3-mediaqueries/` [cited 2018-07-01].

[WAI] World Wide Web Consortium (W3C). *Web Accessibility Initiative (WAI)*. URL: `https://www.w3.org/WAI/` [cited 2017-01-04].

[WAI14] Screen Reader User Survey 5 Results, 2014. URL: `http://webaim.org/projects/screenreadersurvey5/` [cited 2017-01-08].

[WAI17] Screen Reader User Survey 7 Results, 2017. URL: `https://webaim.org/projects/screenreadersurvey7/` [cited 2018-06-16].

[Wat09] Takayuki Watanabe. Experimental evaluation of usability and accessibility of heading elements. *Disability and Rehabilitation: Assistive Technology*, 4(4):236–247, 2009.

[Wir] Wirtschaftskammer Österreich. Barrierefreie Websites - Rechtliche Grundlagen und vorhandene Empfehlungen. URL: `https://www.wko.at/Content.Node/service/t/Barrierefreie-Websites_Inhaltliche-%0AUmsetzung_2015-06.pdf` [cited 2016-10-13].

[WPW12] Petra Winkler, Elisabeth Pochobradsky, and Charlotte Wirl. *Gesundheit und Krankheit der älteren Generation in Österreich*. 2012. URL: `http://www.goeg.at/index.php?pid=produkteberichtedetail&bericht=253&smark=gesundheit+und+krankheit&noreplace=yes` [cited 2016-12-23].

[XFW07] Joseph Xiong, Christelle Farenc, and Marco Winckler. Analyzing tool support for inspecting accessibility guidelines during the development process of web sites. In *International Conference on Web Information Systems Engineering*, pages 470–480. Springer, 2007.

[YHGS03] Yeliz Yesilada, Simon Harper, Carole Goble, and Robert Stevens. Ontology Based Semantic Annotation for Enhancing Mobility Support for Visually Impaired Web Users. In *K-CAP 2003 Workshop on Knowledge Markup and Semantic Annotation*, 2003.

[YSE17] Enes Yigitbas, Stefan Sauer, and Gregor Engels. Adapt-UI: An IDE supporting model-driven development of self-adaptive UIs. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '17, pages 99–104, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3102113.3102144`, `doi:10.1145/3102113.3102144`.

[YSSE17] Enes Yigitbas, Hagen Stahl, Stefan Sauer, and Gregor Engels. Self-adaptive UIs: Integrated model-driven development of UIs and their adaptations. In Anthony Anjorin and Huáscar Espinoza, editors, *Modelling Foundations and Applications*, pages 126–141, Cham, 2017. Springer International Publishing.