

# *Iterative Implementation of a Content Editor for Museum Guides*

Bachelor's Thesis  
submitted to the  
Media Computing Group  
Prof. Dr. Jan Borchers  
Computer Science Department  
RWTH Aachen University

*by*  
*Kevin Fiedler*

Thesis advisor:  
Prof. Dr. Jan Borchers

Second examiner:  
Prof. Dr. Ulrik Schroeder

Registration date: 09.09.2021  
Submission date: 07.10.2021



# Eidesstattliche Versicherung

## Statutory Declaration in Lieu of an Oath

\_\_\_\_\_  
Name, Vorname/Last Name, First Name

\_\_\_\_\_  
Matrikelnummer (freiwillige Angabe)

Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/  
Masterarbeit\* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis\* entitled

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

\_\_\_\_\_  
Ort, Datum/City, Date

\_\_\_\_\_  
Unterschrift/Signature

\*Nichtzutreffendes bitte streichen

\*Please delete as appropriate

### Belehrung:

#### Official Notification:

#### § 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

#### Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

#### § 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Strafflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

#### Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

I have read and understood the above official notification:

\_\_\_\_\_  
Ort, Datum/City, Date

\_\_\_\_\_  
Unterschrift/Signature





# Contents

<b>Abstract</b>	xi
<b>Überblick</b>	xiii
<b>Acknowledgements</b>	xv
<b>Conventions</b>	xvii
<b>1 Introduction</b>	1
<b>2 Related Work</b>	5
2.1 WYSIWYG Content Editors . . . . .	5
2.2 Museum Guides . . . . .	6
2.3 Centre App . . . . .	7
<b>3 Initial Considerations</b>	11
3.1 Stakeholders . . . . .	11
3.2 Centre App Data Structure . . . . .	12
3.3 Current Workflows and Limitations . . . . .	14

---

3.4	List of Requirements	15
3.5	Platform and Code Design Decisions	16
3.5.1	Content Editor	16
3.5.2	Website	18
<b>4</b>	<b>Iterative User Interface Design</b>	<b>19</b>
4.1	Paper Prototypes	19
4.2	Software Prototypes	24
4.2.1	Exhibition Setup Assistant	25
4.2.2	WYSIWYG Style Editing	26
4.2.3	Multi-language Support	29
4.2.4	Audio Player and Keyframes	30
4.2.5	Error Handling	32
4.3	Feature Complete Software Prototype	35
<b>5</b>	<b>User Study</b>	<b>41</b>
5.1	Design and Execution	41
5.2	Task 1: Creating Exhibitions	42
5.3	Task 2: Editing Style	43
5.4	Task 3: Editing Exhibits	45
5.5	Task 4: Audio Player Interactions	48
5.6	Task 5: New Exhibits and Children	49
5.7	Conclusions	51

---

<b>6 Final Product and Conclusions</b>	<b>53</b>
6.1 Finishing Touches . . . . .	53
6.2 Real-World Application and New Features .	54
6.3 Conclusions . . . . .	55
<b>7 Summary and Future Work</b>	<b>57</b>
7.1 Summary and Contributions . . . . .	57
7.2 Future Work . . . . .	58
<b>A Full List of Requirements</b>	<b>61</b>
<b>B Quantifiable User Study Results</b>	<b>63</b>
<b>Bibliography</b>	<b>65</b>
<b>Index</b>	<b>69</b>



# List of Figures

2.1 Map of the Centre app . . . . .	8
2.2 Information content in the Centre app . . . . .	9
3.1 Centre app data structure . . . . .	13
4.1 First paper prototype . . . . .	20
4.2 Second paper prototype . . . . .	21
4.3 Three-pane digital mock-up . . . . .	22
4.4 Two window digital mock-up . . . . .	23
4.5 Exhibition setup assistant . . . . .	25
4.6 NSColorPicker and NSFontPanel . . . . .	27
4.7 Style user interface . . . . .	28
4.8 Multi-language column design . . . . .	30
4.9 Audio player concepts . . . . .	31
4.10 Keyframe popover . . . . .	32
4.11 Audio player info alert . . . . .	33
4.12 Error indicators in the sidebar . . . . .	34

4.13 Concepts for highlighting erroneous values in the exhibit editor	35
4.14 Website	36
4.15 QuickEdit popover	37
4.16 Map & Style workspace	39
4.17 Exhibit and Metadata workspace	40
5.1 Floating map footer	44
5.2 Different appearance dialog	45
5.3 Chart of different workspace interactions	46
5.4 Separate map thumbnail row	47
5.5 Chart of different keyframe interactions	48
5.6 Multiple exhibit selection	51

# Abstract

Many museums offer a mobile app to its visitors these days. These apps are designed to guide the visitors through the museum, list available exhibits, and provide additional information through the app. The benefits for the visitors are manifold. However, workflows for creating and preparing content for museum guides are time-consuming and prone to errors. A content editor can speed up and simplify said workflows.

In this thesis, we use an iterative design process to develop and implement a content editor for museum guides. The content editor provides an easy to use interface, automates certain tasks, and shows a live preview of how the content will look like in the mobile app. We evaluate the content editor using usability heuristics and conduct a user study to learn about expected user interactions and to discover usability issues.





# Überblick

Viele Museen bieten ihren Besuchern heutzutage eine Smartphone-App an. Diese Apps bieten eine Reihe an Funktionen: Sie leiten Besucher durch das Museum, zeigen Exponate an und liefern zusätzliche Informationen. Allerdings ist das Erstellen und Aufbereiten von Inhalten für die Museum-Apps zeitaufwändig und fehleranfällig. Ein Content-Editor kann dabei helfen diese Arbeiten zu beschleunigen und zu vereinfachen.

In dieser Arbeit entwickeln und implementieren wir einen Content-Editor für Museum-Apps. Dafür nutzen wir einen iterativen Design Prozess. Der Content-Editor bietet nicht nur eine einfach zu bedienende Benutzeroberfläche, sondern automatisiert auch bestimmte Arbeitsabläufe und zeigt eine live Vorschau der Inhalte an, wie diese auf den Smartphone-Apps aussehen werden. Wir evaluieren den Content-Editor mittels heuristischer Evaluation. In einer Studie lernen wir mehr über die Erwartungen der Nutzer bezüglich eines Museum-Content-Editors und verbessern die Benutzerfreundlichkeit.



# Acknowledgements

Firstly, I want to thank my supervisors, Sebastian Hueber and Oliver Nowak, for all their time and advice during my work on this thesis.

I want to thank Prof. Dr. Jan Borchers and Prof. Dr. Ulrik Schroeder for examining this thesis.

I thank my family and friends for all their support.

Thank you to everyone who participated in the user study. You provided me with invaluable feedback.



# Conventions

Throughout this thesis we use the following conventions.

## *Text conventions*

Definitions of technical terms or short excursus are set off in colored boxes.

**EXCURSUS:**

Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:  
*Excursus*

Source code and implementation symbols are written in typewriter-style text.

`myClass`

The whole thesis is written in American English. We use the plural form for the first person. For unidentified third persons we use the pronoun they/their.



# Chapter 1

## Introduction

In recent years museums have greatly modernized the museum-going experience [Bae et al., 2013]. With the rising popularity of the smartphone, many museums now offer an app for their visitors to guide them through their exhibitions and provide visitors with additional information and multimedia content through the app.

During our studies we had a look at a wide variety of museum apps across multiple countries<sup>1</sup>. We found that all of them offer the same basic features such as a map of the museum with the location of the exhibits shown. When the visitor selects an exhibit additional information is presented. Some museum apps use text and images, others audio or video files, or a combination of them. Those information is often provided in multiple languages to accommodate visitors from all around the world.

Some apps provide more custom features: locating the visitor within the museum<sup>2</sup>, augmented reality content (AR)<sup>3</sup> or virtual reality content (VR)<sup>4</sup>, or even a quiz section to

Many museum apps offer a lot of similar features.

Museum content can be quite complex.

---

<sup>1</sup>Among others: Germany, the United States, the United Kingdom, Italy, Russia, and China

<sup>2</sup><https://apps.apple.com/us/app/flapp-future-lab-aachen-app/id1145415788>

<sup>3</sup><https://apps.apple.com/us/app/artlens/id580839935>

<sup>4</sup><https://apps.apple.com/us/app/ksc-360-expedition/id1129685069>

check if the visitor paid close attention<sup>5</sup>. Hence, museum content follows a certain, basic structure, but individual features vary based on the individual apps. As [Al Takroui et al. [2008]] points out, museum content can also easily get quite complex.

Museum content needs to be edited manually.

Although many museums already have their information content digitally available [Santoro et al. [2007]], this content may not be in the appropriate format to be used by the mobile app directly. For example, museum apps require the data to be in the same format (e.g., JSON, XML, or SQLite) and images to have a maximum resolution in order to speed up download times. This means that existing museum content needs to be edited and formatted in order to work with the app. This can require tasks such as formatting files, resizing images, or renaming files based on certain naming conventions expected by the mobile app.

Manually editing museum content is time-consuming and prone to errors.

This work alone is rather time-consuming and doesn't yet take into account the possibility of human error. Working with complex data increases cognitive load [Chandler and Sweller, [1991]] which can lead to decreased performance and a higher potential for errors: A mislabeled file or a simple syntax error can render the content unreadable by the app. Furthermore, this work is not necessarily a one-time occurrence. Museums with temporary exhibitions have to repeat this process regularly.

A content editor can speed up and simplify content creation.

One solution to these problems is the use of a content editor which supports the user in their workflow. A content editor can not only reduce the required time by automating certain actions formerly done manually, but it can also reduce the cognitive load by providing an easy to use interface and therefore reduce the amount of errors. However, with the complexity of museum data and the amount of customization required, there is no out of the box software solution.

We develop a content editor based on the Centre app.

In this thesis, we develop a "What You See is What You Get" (WYSIWYG) museum content editor for the "Centre Charlemagne Museum"<sup>6</sup>, a museum in the city of Aachen,

<sup>5</sup><https://apps.apple.com/us/app/explorer-amnh-nyc/id381227123>

<sup>6</sup><http://www.centre-charlemagne.eu>



Germany. The Centre Charlemagne Museum offers a mobile app for iOS and Android called “Centre Charlemagne – Guide” (hereinafter called Centre app) that guides the user through its exhibitions. The Centre app offers a lot of the features mentioned before: A map with exhibits on it and in-depth information for each exhibit in both text form and as an audio guide. Furthermore, the information is provided in multiple languages. The permanent exhibition, for instance, is available in German, English, French, and Dutch.

In Chapter 2 “Related Work”, we have a look at related literature. This includes research into content editors and museum guides. We will also have a detailed look at the Centre app that the content editor is going to be based on.

In chapters 3 to 6 we describe the development process of the content editor. We start by outlining the project, the requirements and initial planing in Chapter 3 “Initial Considerations”. In the next chapter, “Iterative User Interface Design”, we describe the iterative design process of the user interface up to a first feature complete software prototype. We then evaluate this software in Chapter 5 “User Study” and adjust the software based on our findings. In Chapter 6 “Final Product and Conclusions” we introduce the final version of the software and its first use in real-world conditions and present our conclusions of the design process.

In Chapter 7 “Summary and Future Work”, we conclude our research and give an outlook of how the content editor can be expanded in the future.



## Chapter 2

# Related Work

In this chapter, we discuss research that is related to our work. In Chapter [2.1](#), we look into research regarding WYSIWYG content editors and their advantages and disadvantages. We then discuss research into museum guides in general in Chapter [2.2](#), before we look specifically at the Centre app that our content editor is based on in Chapter [2.3](#).

### 2.1 WYSIWYG Content Editors

Research into WYSIWYG editors dates back decades and originates from word processing software [[Chamberlin, 1987](#)]. In fact, the first WYSIWYG word processing software, Bravo, was released in 1974. In the 1990s, research into WYSIWYG also shifted towards web design with WYSIWYG editors for HTML and CSS.

[Williams and Wilkinson \[1994\]](#) developed an WYSIWYG HTML editor and name two benefits over manually editing the document: It is easier to use and guarantees a correct output of HTML documents. However, as [Spiesser and Kitchen \[2004\]](#) point out, the output of WYSIWYG HTML editors can be quite large and contain a lot of unnecessary or repetitive code.

WYSIWYG originates from word processing software.

A WYSIWYG editor guarantees correct output.

A WYSIWYG editor reduces maintenance and development time.

[Chester and Sánchez-Ruíz](#) tested a web based WYSIWYG CSS editor with participants who had experience in HTML and CSS. 83.3% stated that using the WYSIWYG editor offers an advantage over manual editing. They concluded that using WYSIWYG can be beneficial for experimentation and creativity, and that it reduces maintenance and development time.

WYSIWYG lowers the required knowledge.

[Wolber et al.](#) [\[2002\]](#) applied the WYSIWYG approach to dynamic web pages and database access. Using the WYSIWYG interface doesn't require SQL knowledge or any programming of the user. They, too, concluded that using the WYSIWYG system reduces development time and also allows non-programmers to create dynamic web pages.

Similarly, [Yang et al.](#) [\[2008\]](#) used WYSIWYG to simplify the development of web applications with a database based on Social Networking APIs; thus allowing users without programming or database knowledge to develop applications in a graphical user interface. The application generates the database schema automatically and gives users instant feedback on what they have created.

WYSIWYG is also successfully used in specialized areas to accommodate novices and experts alike.

The previous examples of WYSIWYG software were targeted at a broad audience of users. Word processing software such as *Microsoft Word* or *Pages* is probably used by everyone at some point. However, WYSIWYG software can also be used to great success in more niche situations. [Jenny et al.](#) [\[2010\]](#) used a WYSIWYG interface for geospatial data collections. Instead of two types of interfaces, one more suitable for novices and one more suitable for expert users, the WYSIWYG interface met the needs of both groups. By using a WYSIWYG interface, it encourages the user to explore the data. The interface is also flexible, allowing the user to choose their actions in any order.

## 2.2 Museum Guides

There is a lot of research in regard to museum guides. Museums are interested in creating more engaging experiences for their visitors [\[Al Takroui et al., 2008\]](#).

One aspect of research into museum guides is to provide context-aware information. Museum apps can provide context-aware information in different ways such by the user scanning QR codes, using visual recognition, or by tracking the users location [Wein, 2014]. Location tracking allows the user to move freely and still be presented with relevant information [Lanir et al., 2011]. Location tracking can be achieved in many different ways. [Zafari et al., 2019] provides a detailed comparison of different wireless technologies that can be used for tracking such as Wifi, Bluetooth, and RFID.

Museum guides can provide context-aware information.

Context-aware information is not limited to location, though. It can also use personal or environmental factors such as the visitor's interests or the time of day [Cheverst et al., 2000]. In their *GUIDE Project*, a guide for the city of Lancaster, users are able to create custom tours based on selected points of interest (POI).

Context-awareness uses features such as location or visitor's interests.

Another aspect of research concerns new ways of delivering multimedia content to the visitors. Even before it became widely available with smartphones, researchers explored augmented reality (AR) as a way to deliver content. For instance, [Vlahakis et al., 2002] used a laptop and head-mounted display to show a rendered model of the original building over Greek ruins.

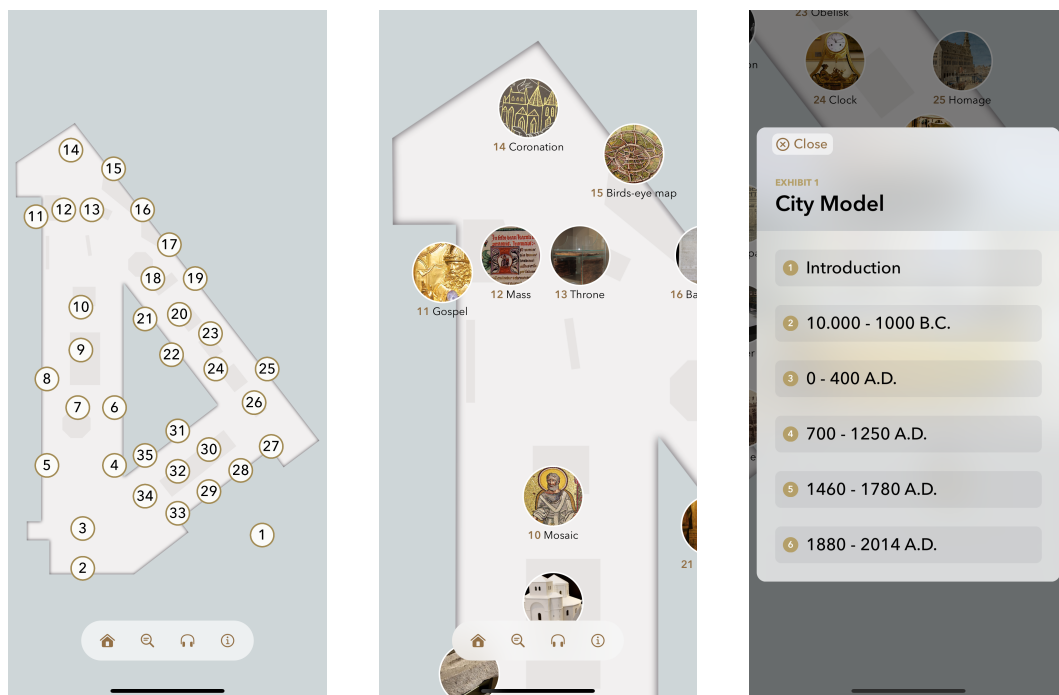
Augmented reality is a modern way to present museum content.

[Ding et al., 2017] provide a view into AR being used in museum apps. Using a smartphone for AR content feels natural to visitors because they are already using their smartphone to take photos. They also look into different types of museums and their respective use of AR: providing additional information for artwork, bringing skeletons back to life, or showing translations for exhibits.

Using a smartphone for AR feels natural.

## 2.3 Centre App

Our museum content editor is build for the Centre app and its set of features. As we have already mentioned, a lot of these features can be found in museum apps all around the world.



(a) Small (zoomed out) map of the museum.

(b) Large (zoomed in) map of the museum.

(c) An exhibit with child exhibits.

**Figure 2.1:** The map of the Centre app. Here, visitors can browse and explore all the available exhibits and their locations.

The Centre app provides content in multiple languages.

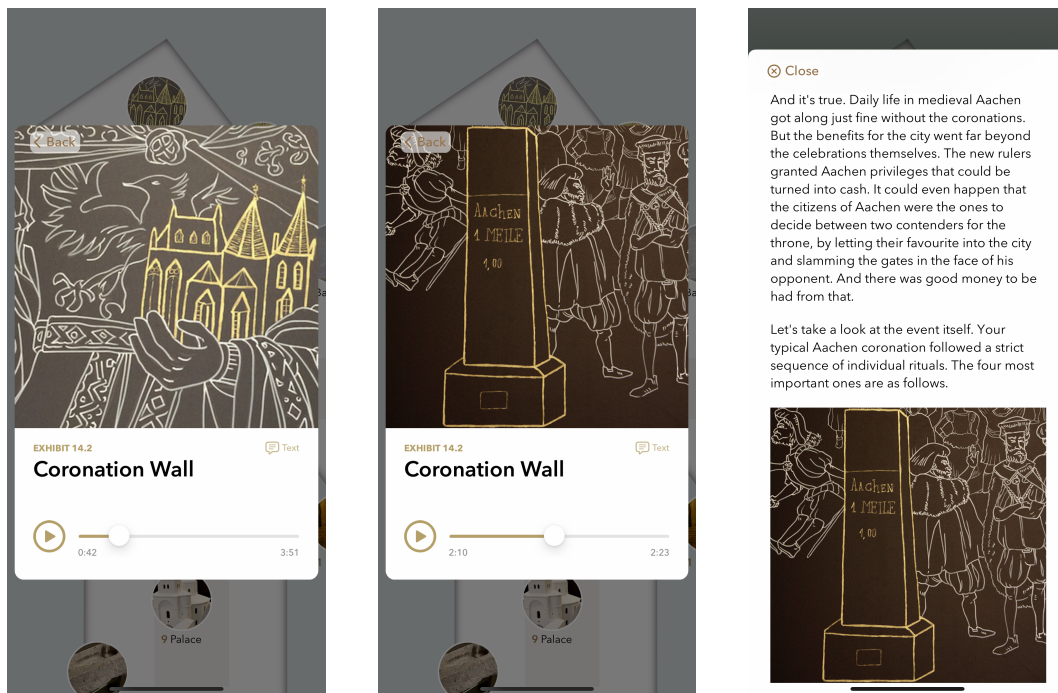
When a user opens the Centre app for the first time, they are presented with an info screen asking them to select their preferred language. The app is providing content in German, English, French and Dutch, although this can vary for individual exhibitions.

Users can explore exhibits on an interactive map.

When the user opens an exhibition, the app shows a map of the museum floor with exhibits on it. The user can zoom in and out of this map. Figure 2.1a shows the zoomed out map (hereinafter small map) and Figure 2.1b shows the zoomed in map (hereinafter large map). Some exhibits might be hidden on the small map and are only visible once zoomed in.

Exhibits can have child exhibits.

Some exhibits have child exhibits (Figure 2.1c), for instance if two exhibits are too close to each other to be shown as single exhibits.



(a) Audio guide interface.

(b) Audio guide interface with a different image after a certain timestamp.

(c) Alternative text content instead of audio guide.

**Figure 2.2:** Each exhibit presents its information either as an audio guide (a,b) or as text content (c).

The Centre app presents content in one of two ways. Firstly, it offers an audio guide (Figure 2.2a). The audio guide shows an image on top, which can change during playback to match the spoken content. Figure 2.2b still shows the same audio guide but at a later time with a different image reflecting the current content. Alternatively, content is provided in text form (Figure 2.2c). In that case, images are displayed within the text content.

The content is available as an audio guide or text.





## Chapter 3

# Initial Considerations

In this chapter, we have a look at the stakeholders involved in the project, the situation and workflow without a content editor, and the requirements for the final product. We will also have a look at code design decisions that are used throughout the design process.

### 3.1 Stakeholders

There are two main stakeholders involved in the development and content creation for the Centre app:

1. There are the **people at the museum** who provide the museum content for the app and design mock-ups of how the exhibitions should look like. This includes designers, editors, translators, photographers, and record artists.
2. There is the **app development team** who develops the mobile apps, formats new content and integrates it into the app.

The people at the museum provide the content.

The app development team adds the content to the app.

Content between both stakeholders is exchanged via email. The people at the museum send new content to the app

Content is exchanged via email.

development team. This content consists of images and a single text document containing the text content for all exhibits in a single language. This also includes an image of the map with locations of the exhibits drawn on it as well as design mock-ups for the style of the exhibition.

Content exchange between both stakeholders is inefficient.

The app development team incorporates the content into the app and applies the styling based on the mock-ups. We have a detailed look at this workflow in Chapter 3.3. Once the new content is added, the development team sends screenshots back to the people at the museum for approval. However, positioning exhibits based on a drawing is not really precise or some titles can be too long in certain localizations. Thus, the fine-tuning requires a lot more communication back and forth.

## 3.2 Centre App Data Structure

Before we can describe the workflows for adding and editing content, we need to have a look at how the Centre app works and the data structures that it requires:

The data structure is identical for iOS and Android.

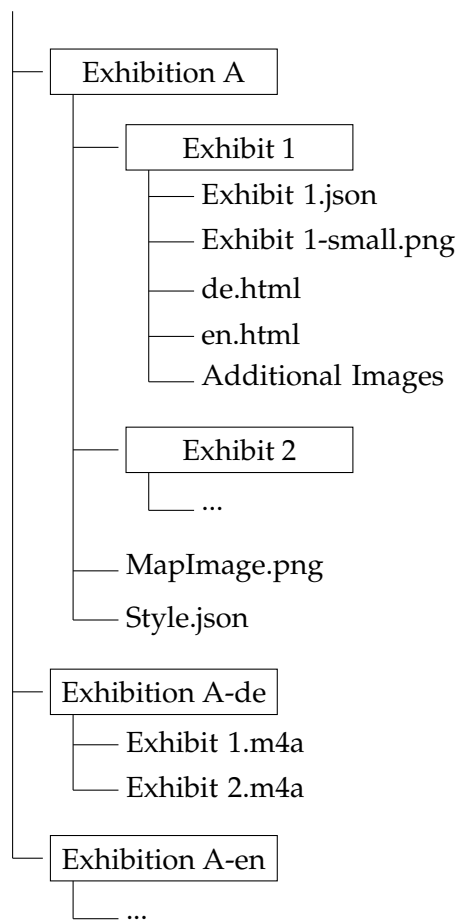
The Centre app uses the same data structure for iOS and Android that is illustrated in Figure 3.1. Each exhibition has one folder containing all the exhibits and required files (map background image, Style.json) and separate folders with a language suffix containing all the audio files for this language. This separation makes the initial download faster. At runtime the user can choose to download the audio files for their preferred language only.

The Centre app uses a multi-level folder structure containing multiple different file types.

Every exhibit inside the exhibition folder has its own folder that contains all the information for this exhibit. There is a `JSON` file containing most of the information (e.g., title, number, coordinates) and an `HTML` file for each language containing the text content. The folder also contains a thumbnail image shown on the map and additional images shown in the audio player or within the text content.

<sup>1</sup><https://www.json.org/json-en.html>

<sup>2</sup><https://html.spec.whatwg.org>



**Figure 3.1:** Simplified files and folder structure used by the Centre app. This sample exhibition contains just two exhibits and supports only two languages.

Furthermore, exhibits can have child exhibits as well. These are the same folders inside an existing exhibit folder. However, the depth is limited to just one; thus, child exhibits don't have children themselves.

Exhibits can have child exhibits.

Exhibit folders are named after an identifier and its JSON file and map thumbnail image use the same identifier as a name (the map thumbnail image adds a "small"-suffix to the identifier). In most cases, this identifier is also used in determining the sort order.

Files and folders have fixed naming conventions.

Exhibition styling is described in a JSON file.

The “Style.json” contains a list of style attributes for the exhibition. This includes information such as colors (as an RGB hex string; e.g., #FF0000 for red), fonts, sizes, and the width of the border. The color attributes can be a single value used in both light and dark appearance or a different values for each appearance.

### 3.3 Current Workflows and Limitations

We observed the app development team to understand the workflows they currently use and, based on that, created a list of requirements (see Chapter 3.4) for our museum content editor. We now have a look at three of these workflows and their limitations:

Manually editing the data structures takes a lot of effort.

The first workflow is about adding and editing exhibits. This firstly requires navigating to the correct folder in a multi-level folder structure. Adding a new exhibit either requires knowledge of all the naming conventions and structure of the JSON file or, more likely, copying an existing folder and changing all the values. Editing the exhibit means editing different files in different formats. The JSON file for general information, HTML files for localized content, image files that need to be resized to a certain resolution. Adding audio files requires looking up the identifier in the JSON file and then adding the audio file with this name to different folders for different languages.

Previewing changes requires to build and run the app.

The second workflow concerns a certain aspect of the exhibit: its position on the map. The position is saved within the JSON file as an  $\{x,y\}$ -coordinate (the top left corner equals  $\{0,0\}$ , and the bottom right corner equals the dimensions of the map image which vary for each exhibition). As we have previously described in Chapter 3.1, the positions of images are provided in form of an image with the positions marked. Based on this image, the app development team has to guess the coordinates of the exhibit and enter them into the JSON file. To verify the coordinates, they have to build and run the app and compare the position within the app with the one provided by the drawing. Then they have to adjust the JSON file coordinate accord-

ingly and repeat this process until the positions match. We took a time measurement of this workflow to highlight its downside: From saving the JSON file, building the app in Xcode<sup>3</sup>, and navigating to the exhibition in the app it took ~12 seconds to just see the results of the changes.

The third workflow is about editing the exhibition style. And it has the same problems as does the positioning. All style attributes are stored inside the “Style.json” and, similar to the positioning, their values have to be inferred from a mock-up image. Getting the colors as an RGB hex string requires a tool such as the “Digital Color Meter” app to analyze the color of a pixel. Here, the app development team often has to further adjust the color because the colors used in print are too bold or too saturated. And this, too, requires that the app is built and run in order to see and verify any changes and compare them to the mock-up.

JSON values have to be inferred from a mock-up.

## 3.4 List of Requirements

Based on our observations of the current workflows (Chapter 3.3) and requests by the app development team, we created a list of requirements for the museum content editor. We now have a look at the key features that the museum content editor should offer. A detailed list of all requirements and features can be found in Appendix A.

We created the list of requirements based on the existing workflows.

- The ability to create new exhibitions and exhibits, including the necessary files and folders.
- A WYSIWYG editor to edit and preview exhibition styling.
- An interactive map that allows to position the exhibits with drag and drop.
- Browsing and editing all exhibits, setting values such as title in all supported languages, adding images and setting up the audio guide.

---

<sup>3</sup>Build time depends on multiple factors such as CPU. We measured with a clean build and the simulator already running on an M1 chip.

- The content editor should be expandable, i.e. designed in a way that allows easy integration of additional features in the future.

Besides the content editor, we also planned for a second piece of software to improve the content exchange between the people at the museum and the app development team. More specifically, we wanted to improve the workflow for exchanging the positions of exhibits on the map.

The website makes fine-tuning exhibit positions faster and easier.

The idea is to build a lightweight website, that allows the positioning via drag and drop. This website is intended to be used by the people at the museum instead of marking the positions of exhibits on an image of the map. Drag and drop not only makes their workflow easier and faster, it also provides the app development team with a precise position for each exhibit. Any changes to the exhibits can then be sent back to the app development team via email in JSON format and the content editor offers an import function for this JSON data; thus it eliminates the need to guess exhibit positions based on image markings.

The requirements for this website are the following: it works with any modern browser, it runs without a server, it doesn't require any changes to the browser security, and it should be intuitive to use and robust.

## 3.5 Platform and Code Design Decisions

### 3.5.1 Content Editor

We decided that the platform for the content editor would be macOS with [Swift<sup>4</sup>](https://developer.apple.com/swift/) as the programming language. In contrast to the mobile apps, which are designed to run on a large number of devices and thus have to support older operating systems (iOS 9 and Android 6) as well as the most recent ones, the content editor does not need this backwards compatibility.

<sup>4</sup><https://developer.apple.com/swift/>

The typical design pattern for an AppKit macOS app is the Model-View-Controller (MVC) pattern. However, given the size and complexity of the app, we also use reactive programming with declarative code to process values over time. This was achieved by utilizing Apple's [Combine framework](https://developer.apple.com/documentation/combine)<sup>5</sup>, a relatively new framework that bundles all the previous asynchronous event handlers (Notifications, Key-value-observation) into one. We use reactive programming mostly to update the UI (e.g., updating the live preview map) when the model changes and by doing so reducing the amount of code in the controller.

The content editor uses MVC and reactive programming.

We also decided to reuse some of the code from the iOS mobile app to make it easier to maintain or add new features to both. This applied to the model objects for exhibit and style. We also tried to use the same names for variables. However, some adjustments were necessary. For instance, the iOS app only needs to read the data and could therefore rely on value types (structs). The content editor on the other hand uses reference types (classes) to allow changes from multiple different views. There were other necessary changes such as editing all languages in the content editor (as opposed to displaying just one in the mobile app) and supporting light and dark appearances<sup>6</sup> at the same time (as opposed to just the current appearance in the mobile app).

We reuse some of the iOS app code to make maintenance easier.

A final decision regarded data storage: The decision was between keeping the folder structure as described in Chapter 3.2 or create a new model in CoreData. Using CoreData would have potentially improved performance a bit, but we decided to keep the mobile app folder structure for two reasons:

The content editor uses the same data structure as the mobile apps.

1. It remains a single source of truth (SSOT), meaning that the model is saved in one location only and is therefore always up-to-date. This also eliminates the need to export the data for every change.

---

<sup>5</sup><https://developer.apple.com/documentation/combine>

<sup>6</sup>When we talk about light and dark appearance, we refer to the depicted appearance of the WYSIWYG part of the content editor and not the appearance of the content editor itself.

2. It still allows for manual edits if required. It also allows potential scripts to batch edit a lot of files at once.

Working on the original data meant that we had to put special emphasis on write safety as not to corrupt the data by accident and build in measures to prevent errors (Nielsen's fifth usability heuristic) or recover from them [Norman, 2013] with *undo* capabilities implemented throughout the editor.

### 3.5.2 Website

The website is dynamic and generated at runtime.

The website is supposed to be small and lightweight. This meant that we designed the website in HTML5, CSS and JavaScript without using any third-party libraries. The idea was to use the HTML `<template>`-tag to create the website dynamically from JavaScript at runtime with the data provided by a JSON file.

However, there is a problem with accessing a local JSON file, even in the same folder, with the browser default Cross-Origin Resource Sharing (CORS) restrictions. Instead we use a JavaScript variable containing the JSON data as a single string. This variable along with exhibit thumbnail images are all generated by the content editor.

We discuss the design of the website in Chapter 4.3.



## Chapter 4

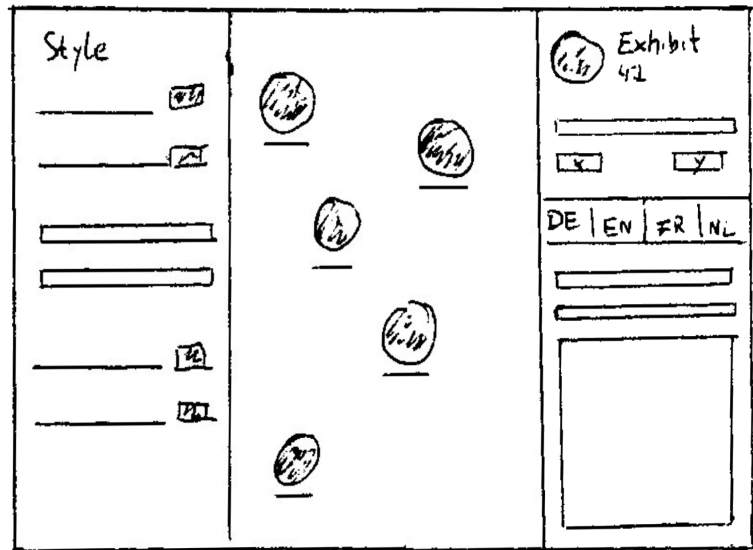
# Iterative User Interface Design

We developed our museum content editor with a human-centered design process [Norman, 2013] and worked closely with the app development team to ensure that all requirements are met and that usability is high. The design process is iterative following a DIA cycle (Design - Implement - Analyze) and with each iteration we refined the user interface (UI) or parts of it.

In Chapter 4.1, we describe initial design concepts in form of paper prototypes and digital mock-ups. These designs are focused on the UI as a whole. In Chapter 4.2, we continue with a higher fidelity horizontal software prototype and focus on some individual elements of the UI. Finally, in Chapter 4.3, we describe the first feature complete software prototype that we evaluate in a user study in Chapter 5.

### 4.1 Paper Prototypes

We started the design process by analyzing our list of requirements (Chapter 3.4) and came up with three elements that the content editor would need:



**Figure 4.1:** A paper prototype of the initial three-pane window design: style (left), map (center), and exhibit (right).

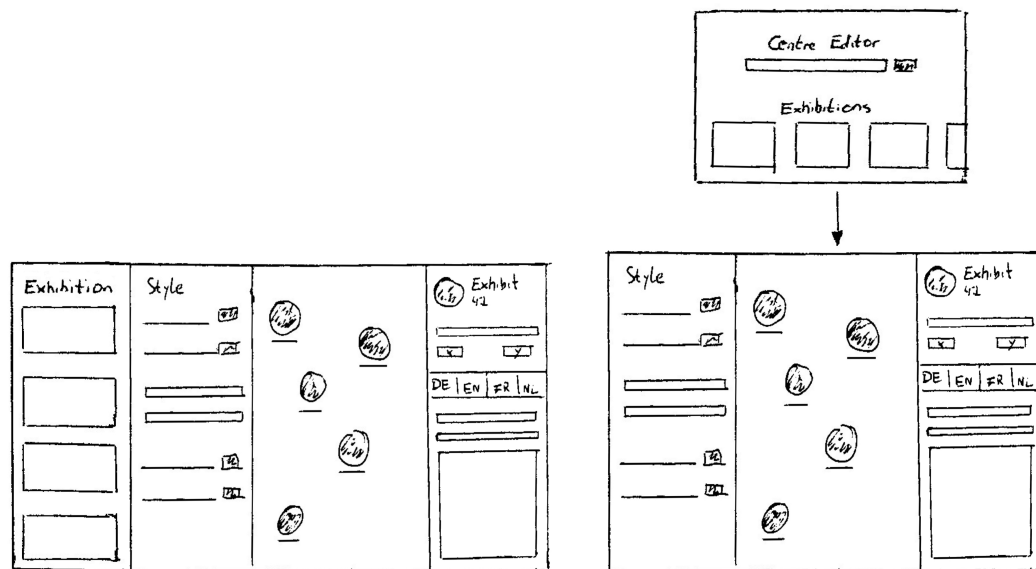
1. A front-end to edit the style JSON file.
2. A front-end to edit exhibits.
3. A live preview of the map, showing the exhibits at their correct positions and reflecting the changes to the style.

The initial paper prototype design uses a single three-pane window.

As an initial design we created a three-pane window concept with each pane containing one of the three main features: the style attributes on the left, the map (with WYSIWYG) in the middle, and the exhibit on the right. Here we considered showing a single exhibit selectable from the map or a scrollable list of all exhibits. This design as paper prototype is shown in Figure 4.1.

We had two concepts for browsing exhibitions: a fourth pane or a welcome window.

We then analyzed how users would interact with this design. We had a list of tasks based on the former workflows and analyzed how users would perform them in this prototype. Early on, we discovered one shortcoming: The design only shows a single exhibition and lacks the option to browse all the available exhibitions. We came up with two solutions as shown in Figure 4.2. The design on the left (a)



(a) Four-pane window with exhibitions in the left sidebar.

(b) Separate window to browse exhibitions and original three-pane window.

**Figure 4.2:** Two concepts of how to handle exhibition browsing: four-pane window (a) and welcome window opening the three-pane window (b).

uses a fourth pane at the left most position to browse and select exhibitions. The design on the right (b) shows a separate window (“welcome window”) when the app launches where the user can select the exhibition. The selected exhibition then opens in the previously designed three-pane window.

We compared the two designs with each other. While a single window would be less cluttered and possibly reduce mouse travel times, we decided against it and chose the *welcome window* concept instead. We assume that a user would typically only work on a single exhibition (e.g., a new temporary exhibition). Therefore, a fourth pane that is interacted only ones at launch would take up a lot of screen space despite being used little to none. Based on this, we also decided to provide the option to skip the *welcome window* altogether and launch directly into the newest exhibition.

We decided to use a skippable welcome window.



**Figure 4.3:** The three-pane concept with *welcome window* as digital mock-up created in *Affinity Photo*.

The *welcome window* has another advantage: apps only have access to certain files and directories when using [App Sandbox<sup>1</sup>](https://developer.apple.com/documentation/security/app_sandbox/), a default security measure. In the *welcome window*, the user can select the content directory and thereby grant the app permanent access to the directory.

We decided to continue with a higher fidelity at this point and created a mock-up of our paper prototype in *Affinity Photo* to get a better understanding of how the user interface would actually look like. This design mock-up of the *welcome window* and the three-pane exhibition window is displayed in [Figure 4.3](#).

The exhibit content is too extensive to be placed in a sidebar.

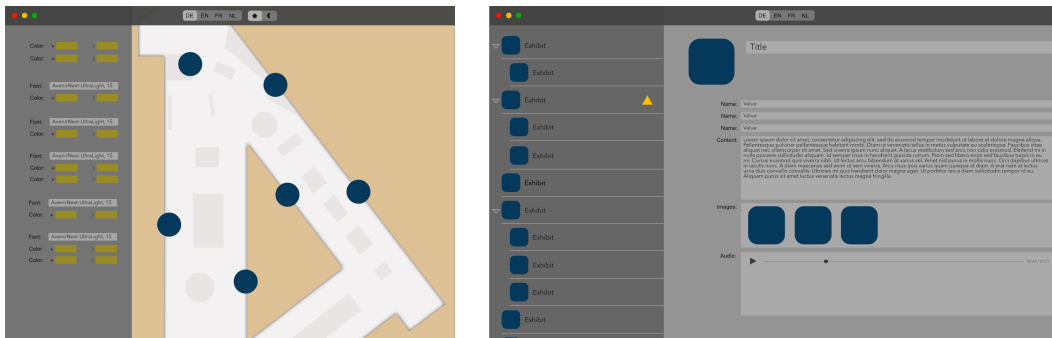
However, this revealed a major flaw in our design concept: The content for each exhibit is way too extensive to be placed in, what effectively is, a sidebar. It also made it unnecessarily difficult to visualize the tree-like exhibit structure that we described in [Chapter 3.2](#).

We created a second window with an outline view and exhibit editor.

We removed the exhibit sidebar and thus turning the formerly three-pane window into two panes with the style in the first pane and the interactive map in the second pane. And we designed a second window with an [outline view<sup>2</sup>](#) on the left and an editor on the right (see [Figure 4.4](#)).

<sup>1</sup>[https://developer.apple.com/documentation/security/app\\_sandbox/](https://developer.apple.com/documentation/security/app_sandbox/)

<sup>2</sup><https://developer.apple.com/documentation/appkit/nsoutlineview>



**Figure 4.4:** A mock-up of separate windows for different workflows. The window on the left shows the style and map. The window on the right shows the exhibit tree structure and the exhibit content editor.

#### OUTLINE VIEW:

An outline view is a UI element for displaying tree structures. Children of parent nodes are indented and parent nodes have a disclosure triangle to collapse or expand their children.

Definition:  
*Outline View*

The outline view allows the user to browse all the exhibits hierarchical and upon selecting one it is editable in the right part of the window. This design has the additional benefit that the user can focus on a single exhibit when editing without getting overwhelmed by too much data. This conforms greatly with [Nielsen's](#) eighth heuristic that the user interface should only display information relevant for the current task.

While this solved the issue of a cramped user interface we did not like the idea of having two separate windows for data within the same context. Both windows belong to the same exhibition and work on the same data so it didn't make sense to separate them [[Lauesen and Harning, 2001](#)]. We experimented with the concept of “workspaces” by adding a popup button to the right side of the toolbar. That way, we went back to a single window that would change the entire user interface based on the current workflow. This is something that other applications, that offer a lot of different tools and workflows (e.g., photo or video editing software such as *Affinity Photo* or *Adobe Lightroom*), do as well.

We merged the two windows into one by using workspaces.

Workspaces have lower interaction costs than multiple windows but slightly worse discoverability.

The benefits are that this way the same data is only associated with a single window and by implementing meaningful keyboard shortcuts (we decided on  $\text{⌘} + \text{1}$ ,  $\text{⌘} + \text{2}$ ) changing workspaces would be a lot faster and require less interaction costs [Budiu, 2013] than selecting a different window. The workspaces have the same advantages that we had with multiple windows, i.e., allowing the user to focus on a single workflow at the time by removing unnecessary UI elements and thus removing distractions. It also makes the editor easier to expand in the future if new features should be added. The only downside is that discoverability might suffer; something that we tested during our user study (Chapter 5.4).

We simulated the workspace behavior with multiple layers inside *Affinity Photo*. However, at this point it became more and more important to actually be able to interact with the prototype that we decided to continue with a horizontal software prototype based on this design.

## 4.2 Software Prototypes

We created an initial horizontal software prototype in [Xcode](#)<sup>3</sup>. We also implemented the read function for the exhibition data in order to display real values in the user interface instead of placeholders. This made it a lot easier to evaluate the UI and adjust sizes and margins based on the data. The project file and all the code is available [here](#)<sup>4</sup>.

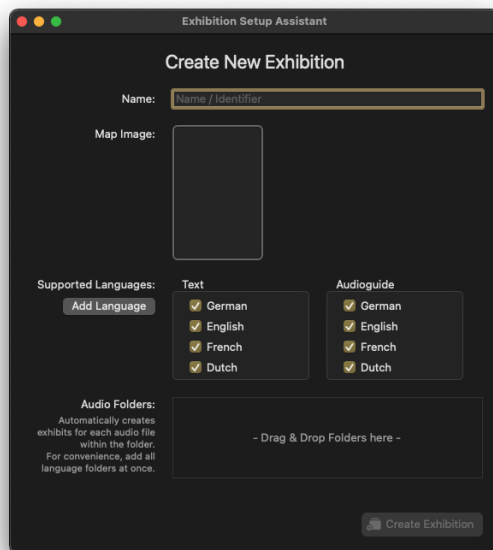
When building the user interface, we made sure to only rely on standard UI elements so that the user interface feels familiar. We also followed the [Human Interface Guidelines](#)<sup>5</sup> so that the content editor matches the expected macOS look and design. For icons we use [SF Symbols](#)<sup>6</sup>, a library of icons designed by Apple to work seamlessly with the system font.

<sup>3</sup><https://developer.apple.com/xcode/>

<sup>4</sup><https://git.rwth-aachen.de/i10/Centre-Editor>

<sup>5</sup><https://developer.apple.com/design/human-interface-guidelines/macos/overview/themes/>

<sup>6</sup><https://developer.apple.com/sf-symbols/>



**Figure 4.5:** The design for the exhibition setup assistant. Only a name is required to create a new exhibition, everything else is optional for a more convenient setup process.

In the next part of this chapter we have a detailed look at some particular aspects of the UI and how we refined them.

### 4.2.1 Exhibition Setup Assistant

When creating a new exhibition the only requirement is a name that is used to create the exhibit folder and audio folders. However, we created a wizard-style assistant, too. Figure 4.5 shows our design of the *Exhibition Setup Assistant*. In addition to the required name text field, users can add the map background image, select the available languages and batch import exhibits based on audio files. All these additional inputs are optional and can also be added and changed from within the exhibition window.

The exhibit batch import is a convenience feature that makes the initial setup a lot easier and faster. For exhibitions with audio guide, there is usually an audio file for

We created a setup assistant for creating new exhibitions.

The assistant can automatically create exhibits for all audio files.

each exhibit. When the user chooses to import this folder containing the audio files, the setup assistant can automatically create a new exhibit for each audio file and arrange them in a grid pattern on the map.

There are certain usability issues with wizards such as higher interaction costs and potentially blocking the app due to modal windows [Budiu, 2017]. Those issues don't really apply to our exhibition setup assistant.

Firstly, the creation of a new exhibition is a relatively rare occurrence. Thus, even users familiar with the content editor can benefit from additional guidance through the process. The assistant is also not modal and does not block the remainder of the content editor. Lastly, apart from the name all consecutive steps are entirely skippable.

## 4.2.2 WYSIWYG Style Editing

Style attributes, just as exhibit information, is stored in a JSON file. For both, we use the same concept to work with the files: Firstly, we use a `Codable`<sup>7</sup> private struct that is modeled after the JSON file contents and handles reading from and writing to the file. However, as we have already mentioned in Chapter 3.5.1, a struct is a value type and we need a reference type (class) in order to use Combine `Published`<sup>8</sup> properties. In case of the style, we therefore use a class that uses the `Codable` struct internally to initialize and to save changes to the JSON file. This class conforms to the `ObservableObject`<sup>9</sup> protocol which means that it can notify the user interface if any of its properties change. Each style property uses the `@Published` property wrapper which in turn can notify the user interface if this particular value changes.

We use Combine to effortlessly update the UI.

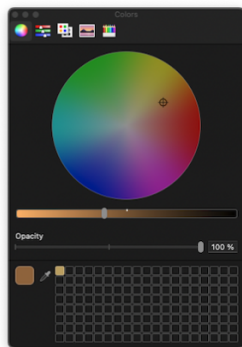
The advantage of this approach is that with very little code we are able to update the model and update every part of the user interface that is affected by the change. This is also

<sup>7</sup><https://developer.apple.com/documentation/swift/codable>

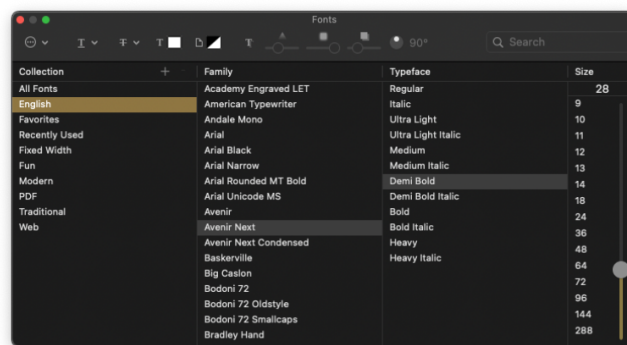
<sup>8</sup><https://developer.apple.com/documentation/combine/published/>

<sup>9</sup><https://developer.apple.com/documentation/combine/observableobject>





(a) NSColorPicker



(b) NSFontPanel

**Figure 4.6:** macOS provides native UI elements to edit colors (a) and fonts (b).

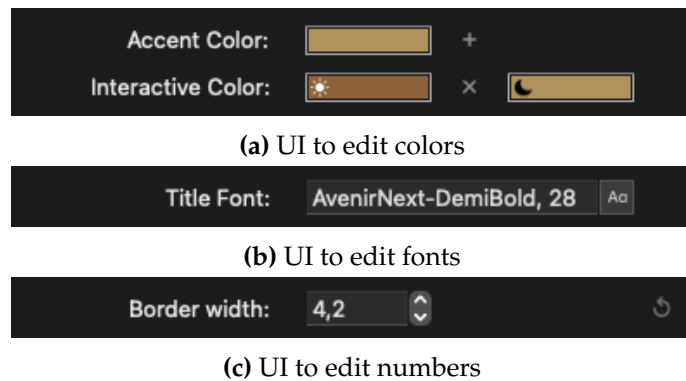
especially useful if new style attributes were to be added (see Chapter 6.2). The second advantage of the `Style` class is that we cast the optional values from the JSON file into their proper types and provide default fallback values if some are missing.

With the use of `Combine`, this code is fully capable of handling WYSIWYG. Any change is instantly applied to the user interface and thus provides a live preview. We now look at the design of the user interface and our attempts to make editing the style more natural and convenient.

Style attributes are mostly one of three types: Colors represented by an array of `String` (one or two elements), fonts represented by a `String` containing the name of the font and its size, and numbers either as `Int` or `Float`.

A color can either be a single color used in both light and dark appearance or it can have one separate values for each appearance. To edit color, we use the native `NSColorWell` component, that displays the current color and shows the system provided `NSColorPicker` (Figure 4.6a) on click where the color can be changed. The color picker includes a pipette tool to sample colors from any pixel on the screen, thus making the “Digital Color Meter” app no longer necessary.

The control for editing color uses the native `NSColorWell` and `NSColorPicker`.



**Figure 4.7:** The different user interface elements to edit the style. If a value varies from the default style (c) a reset option is displayed on the right.

To use different colors for different appearances, we created a user interface as shown in Figure 4.7a. If a single color is used for both appearances (in this case the accent color), only a single color well is shown. The user can add a dark appearance variant by clicking the plus button. With different colors for each appearance (here the interactive color), two color wells are shown. They include a little icon to indicate their respective appearance.

The control for editing fonts uses a text field and the `NSFontPanel`.

To edit fonts, we use a text field with the name and size of the font (Figure 4.7b). We also added a button to open the system `NSFontPanel` (Figure 4.6b), where the user can browse all available fonts.

The control for editing numbers uses a text field and a stepper.

To edit number values, we use text fields together with steppers (Figure 4.7c). Based on the value in question, they increase or decrease in appropriate steps. They can also be adjusted with `↑` and `↓` or by scrubbing their label for quick adjustments. The latter are accelerators (Nielsen's seventh usability heuristic) that can speed up the workflow for advanced users.

All these design decisions are meant to simplify editing the style. For instance, it is no longer necessary to find out the exact *hex* value of a color. All this complexity is hidden away under familiar UI elements and handled by the content editor instead.

### 4.2.3 Multi-language Support

Unlike the Centre app, which only shows information in one selected language, the content editor has to edit content in all supported languages. We solved this from a code point of view by implementing a `generic`<sup>10</sup> struct that wraps the content and its associated language: `struct LocalizedContentWrapper<T>` works as a getter and setter for `class LocalizedContent<T>` which contains the associated language and the generic content. This content is the same as in the mobile app and mostly `String` or `URI`<sup>11</sup> but in case of the audio guide it is even an `NSObject` class that contains images and their timestamps.

Originally, we used an `enum` for the language but during an evaluation with the app development team it became apparent that hard-coding the languages is contrary to our “expandable” requirement and that there are indeed use cases where adding additional languages at runtime would be beneficial. So instead, we used a JSON file where languages are read from and written to and which is entirely editable at runtime.

In our initial design (three-pane window) we had only considered the UI to show a single language. The map and the exhibit sidebar would be the same language and users could change the language from a segmented control in the toolbar. With the switch towards workspaces, this limitation was no longer necessary. We explored the possibility of showing all languages in the exhibit content editor at once in column views (Figure 4.8) as opposed to the single language view (Figure 4.4). The design is similar to a table with languages in each column and data in each row.

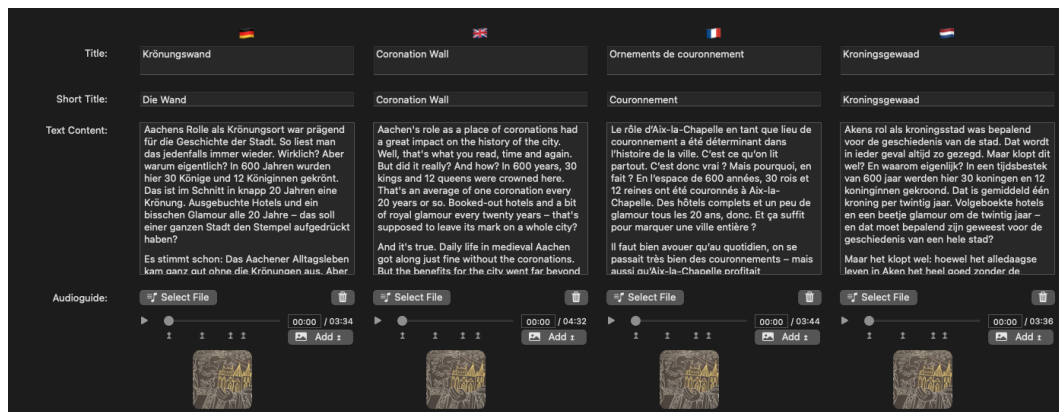
We evaluated both design concepts with the app development team. Displaying a single language at a time has the advantage of a cleaner UI and better focus. Displaying all languages in columns has the advantage that all data is visible at the same time and that it is easier to spot missing data. It also allows for two different workflows: filling out

Languages are stored in a JSON file and editable at runtime.

We designed a UI for the exhibit editor where all languages are visible.

<sup>10</sup><https://docs.swift.org/swift-book/LanguageGuide/Generics.html>

<sup>11</sup>URL refers to local file urls.



**Figure 4.8:** A concept for showing data in all available languages. Each language is in its own column.

a single language (vertical) or filling out a single value in each language (horizontal). We decided to use the multi-language column design by default, but added an option in the settings to switch to single language mode.

#### 4.2.4 Audio Player and Keyframes

The audio guide with its changing images is one of the key features of how a museum visitor interacts with the Centre app. Getting the timing of the images right is important and we explored different designs and interactions that best allow setting and fine-tuning [keyframes](#)<sup>12</sup>.

Definition:  
*Keyframe*

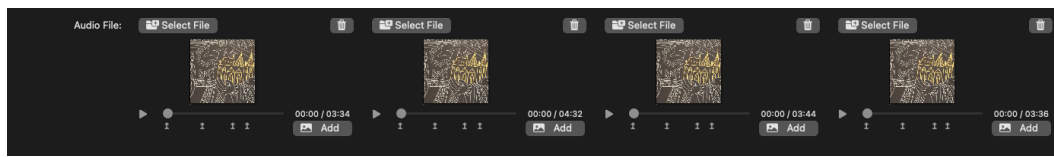
##### KEYFRAME:

Keyframes are used in animation and filmmaking software for defining start and end points for transitions. For the audio guide, a keyframe refers to the start point (timestamp) of a new image.

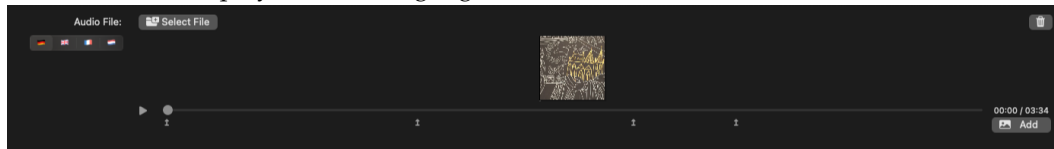
We tested if a longer timeline is beneficial for the audio player.

With the multi-language column design, that we introduced in Chapter [4.2.3](#), the length of the timeline became rather short. In Figure [4.9](#) we tested if a full-width timeline would be a better choice. However, it didn't make sense

<sup>12</sup>[https://en.wikipedia.org/wiki/Key\\_frame](https://en.wikipedia.org/wiki/Key_frame)



(a) The audio player for all languages within the columns of the localized content.



(b) The audio player for a single language with a longer timeline.

**Figure 4.9:** Two designs for the audio player: on the top (a), keeping the audio player within the language columns; on the bottom (b), having an elongated timeline for a single language.

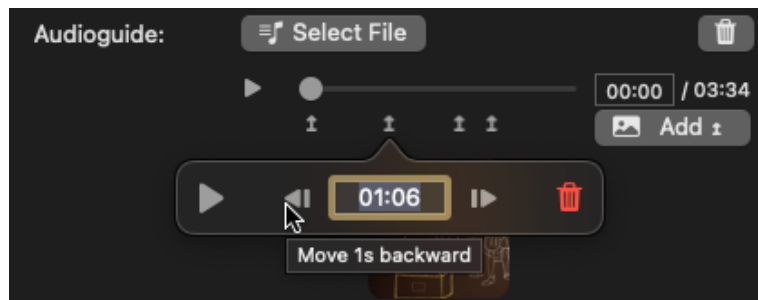
to have part of the localized content in the column view and another part not. Furthermore, keyframes are only set at one-second precision. With a minimum column width of 300px this allows audio files up to five minutes to have keyframes accurate to one pixel. The longest audio file in the permanent exhibition is 4:30 and the average length is about 2 minutes. Although this might be something to revisit if longer audio files are used.

We considered two possible ways to add new keyframes: one is to insert a keyframe at the timestamp where the current playback position is at. We added a popup button for this that shows all available images that can be inserted. The other interaction is that the image is dragged and dropped on the timeline at the position where the keyframe should be created. We considered both interactions very plausible and examined this during our user study (see Chapter 5.5).

Keyframes can be added by button or with drag and drop.

The second kind of interaction with keyframes is changing their location. Here, we assumed that the keyframes are initially placed in the approximate location where they should be. Then, they would only need to be fine-tuned to the audio. So we added a context menu to move them one second forwards or backwards (the precision of timestamps is only to the second).

Originally, keyframes could only be moved in one second increments.



**Figure 4.10:** A keyframe popover where the user can set the timestamp directly or move it in one second increments.

We added drag and drop to keyframes and allowed to set a precise timestamp.

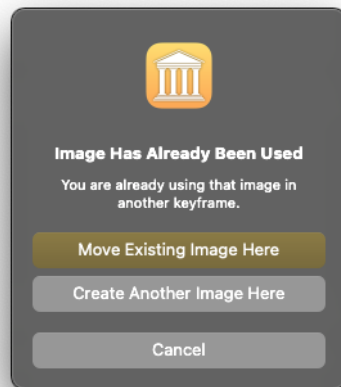
When we tested this with the app development team, our assumption turned out false. Often the initial placement of keyframes was off by multiple seconds and moving them in one second increments was cumbersome. While we still wanted to retain the option to fine-tune the keyframes, we also needed a way to adjust keyframes more quickly. Firstly, we added drag and drop to the keyframes to move them around directly. This allows to perform big adjustments quickly. Secondly, we added a popover (Figure 4.10) accessible from the context menu or via double click. Within the popover, the keyframe can be moved in one second increments, but also directly by entering a timestamp in the text field.

#### 4.2.5 Error Handling

The content editor handles user input constantly. Therefore, it is important that the editor keeps the user informed at all times and handles errors gracefully.

Error dialogs try to offer a solution directly from the alert.

If an error dialog is displayed and the content editor is capable of fixing the problem itself, it will offer its solution directly from the error dialog [Shneiderman et al., 2016]. For instance, when the user adds the same image twice to the audio player (something that is currently not done by any exhibition, but we didn't want to assume that it might not be done in the future) an info alert (Figure 4.11) is shown to the user.



**Figure 4.11:** An info alert by the audio player offering two possible solutions as well as an option to cancel.

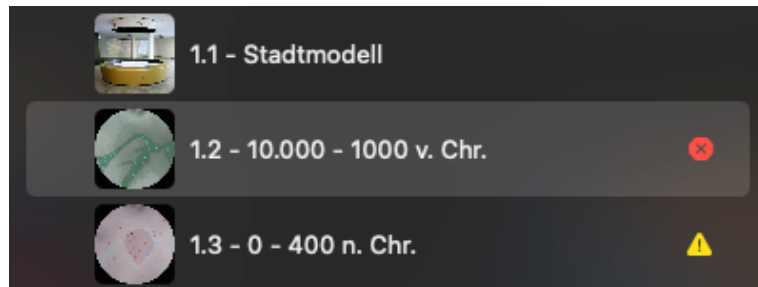
It not only informs the user, that this action might not have been what they wanted, but also offers to move the existing keyframe to this position instead. Similarly, when adding a new keyframe to the same timestamp as an existing one, the user is asked if he wants to change the existing image instead. These dialogs not only offer solutions, but also a cancel option as an exit (Nielsen's third usability heuristic).

In situations where the content editor can't offer a solution itself, it instead informs the user of possible alternatives. For example, if the user tries to add an image in an unsupported format the info dialog lists all the supported image formats.

And if there is an error where the content editor can't offer a solution itself and can't provide alternative solutions, it displays the error message in plain English (Nielsen's ninth usability heuristic).

The previous errors dialogs are used in situations where an error should be fixed right away. We also believed that a second kind of error indication would be useful. Exhibitions often have thirty or more exhibits each with a large

The outline view shows error indicators to quickly identify exhibits with problems.



**Figure 4.12:** The outline view in the exhibit sidebar shows error and warning indicators for incorrect exhibits.

number of values with most of them non-optional. If an exhibit has a missing value or some other kind of mistake (e.g., a different number of keyframes in different languages), an error indicator (a yellow triangle with an exclamation point for warnings and a red octagon with a cross for errors) is displayed directly in the exhibit outline view (Figure 4.12).

This way, the user can quickly identify exhibits with errors and fix them. This kind of visualization is also used in IDE software (e.g., Xcode) and should therefore be familiar to the app development team. However, the error indicator in the sidebar only shows that the exhibit has an error and not what exactly the error is.

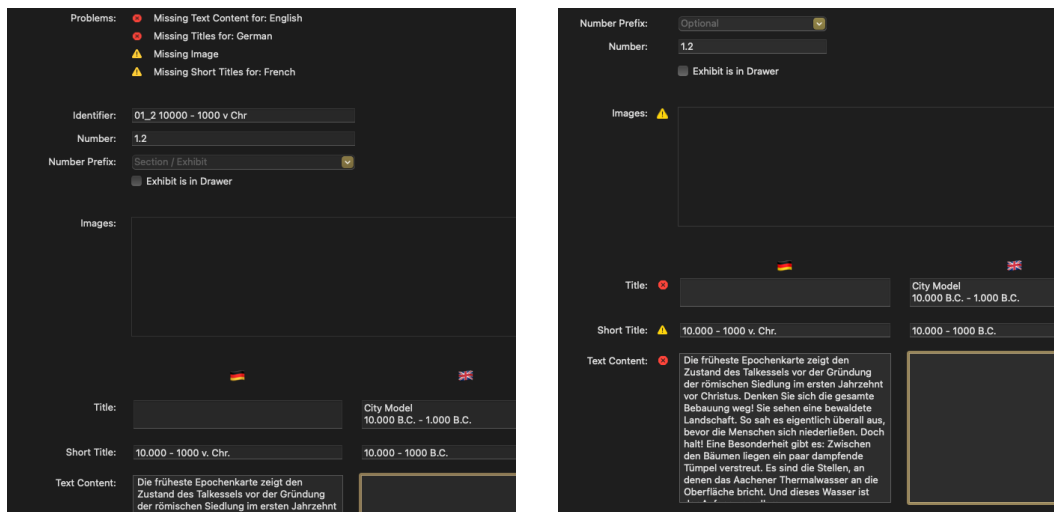
The exhibit editor shows error messages at the top of the window.

To solve that, we came up with two concepts for the exhibit window. The first one (Figure 4.13a) is to show all error messages at the top of the exhibit editor. They are in one prominent spot and have the error message visible at all times. However, if the amount of errors changes, the whole interface shifts a little bit up or down. It is also quite far off between the error message and the value in question.

The exhibit editor shows error indicators next to affected values.

The second concept (Figure 4.13b) solves both issues by displaying the error indicator right next to the value in question. This way, the user will instantly associate the error indicator with the value next to it. The downside is that the error message is not visible. However, the Law of Proximity [Wertheimer, 1938] suggests that items close to each other are perceived as a unified group and thus making the error





(a) This concept shows all error messages at the top of the content editor.

(b) This concept shows error indicators next to two the values with errors.

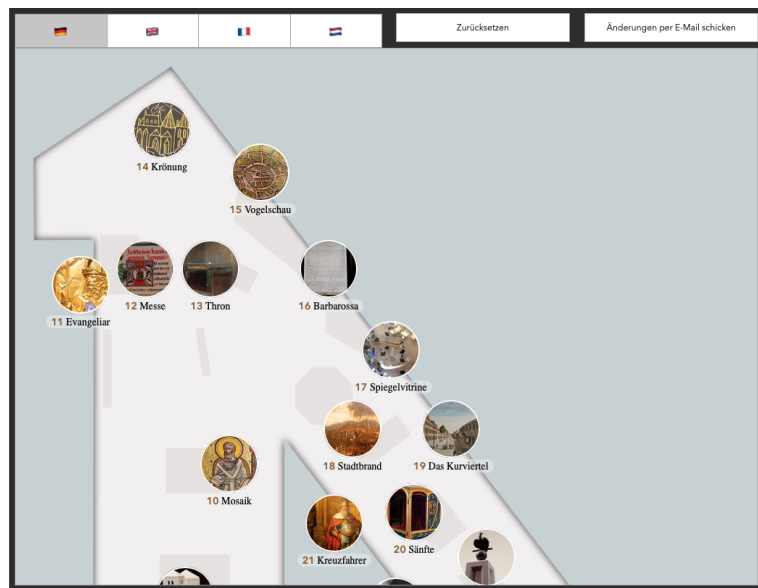
**Figure 4.13:** Two concepts for highlighting wrong or missing values in the exhibit editor: (a) shows all error messages on top, (b) shows error indicators next to affected values.

message unnecessary in most cases. It can still be displayed as a popover by clicking the error indicator though.

Both designs had their advantages and disadvantages. Ultimately, we decided to use the second concept with the error indicators next to the values mainly because the otherwise shifting UI felt weird when experiencing it.

## 4.3 Feature Complete Software Prototype

With the UI working as intended we implemented the remaining missing functions. This included saving the data, certain automations (e.g., resizing images upon import, and generating the map thumbnail image), the addition of a third workspace for metadata and export, and the import and export functionality for the exhibit positioning website.

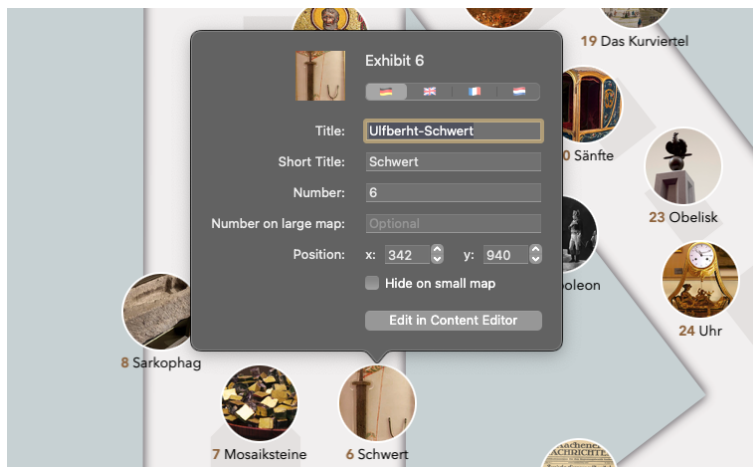


**Figure 4.14:** The content editor exports this website that allows the people at the museum to position the exhibits naturally using drag and drop.

The website is kept simple with focus on tweaking the design on the map.

We kept the functionality of the website simple. Figure 4.14 shows our design. The main part of the website is the map with the exhibits on it. The exhibits can be moved via drag and drop. At the top of the website, all supported languages can be selected and previewed within the map. The website also allows to edit the titles underneath each exhibit simply by clicking on them. If any changes occur, two buttons appear in the top right corner. The left one resets the exhibits to their original values (after confirmation) and the right one generates a JSON string with all the changes and prepares an email with everything already entered (recipient, subject, body) that only needs to be sent.

This resulted in our first feature complete software prototype. At this point, we did a heuristic evaluation of the whole user interface. We specifically checked the former workflows, that we outlined in Chapter 3.3, to verify that they worked as intended.



**Figure 4.15:** The QuickEdit popover allows changing map-related properties directly on the map workspace.

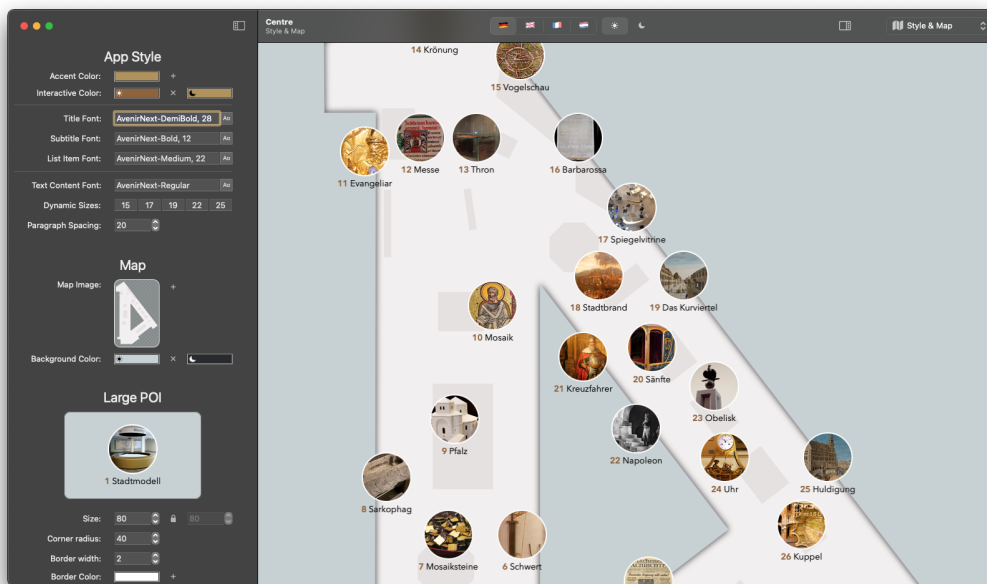
There we discovered one more usability issue: On the *Map & Style* (herinafter *Map*) workspace, users are able to edit the style and position the exhibits. But there they would also check the length of exhibit title and possible overlap issues and whether or not exhibits would need to be hidden on the small map. All these properties were only available in the *Exhibit Editor* (herinafter *Editor*) workspace. Thus, users would need to change the workspace, find the exhibit in question, edit the value and change back to the *Map* workspace to verify the change. Instead, we added a feature to the map that we call *QuickEdit*: a popover that appears over the exhibit either using the context menu or the *Quick Look* gesture (default: three finger tap). In this popover (Figure 4.15) users are able to change all map-related properties of the exhibit and get an instant preview on the map.

We also added some accelerators (Nielsen's seventh usability heuristic): Firstly, we added keyboard shortcuts for most tasks (e.g.,  $\text{⌘} + \text{⌘} + \text{Number}$  to switch between languages, or  $\text{⌘} + \text{D}$  to toggle the current appearance). Secondly, we expanded drag and drop from external sources. Previously, only image files could be dragged onto an exhibit. We added audio files for the audio player and text files for the text content to this.

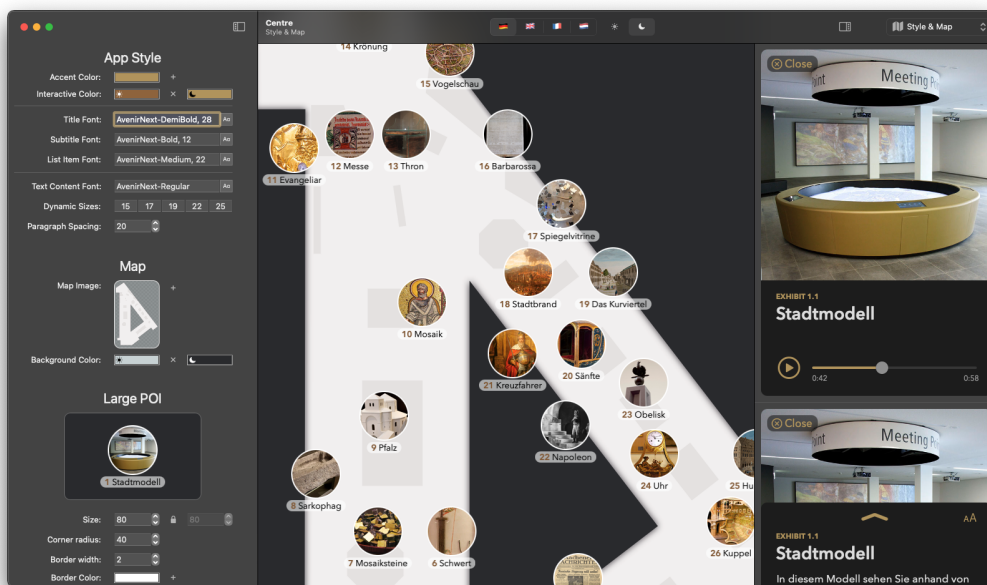
We added a popover to exhibits on the map to quickly change map-related values.

We added keyboard shortcuts for common tasks and expanded drag and drop.

With these changes, the content editor seemed advanced enough that we could evaluate it in a user study. Figures 4.16 and 4.17 show our final design that is used in the user study in Chapter 5. Figure 4.16 shows the *Map & Style* workspace. Figure 4.16a shows the map in light appearance whereas Figure 4.16b shows the dark appearance. On the right side of Figure 4.16b are static previews of the content (i.e., for the audio guide and the text content). Figure 4.17a shows the *Exhibit Editor* workspace with the exhibit list in an outline view on the left and the editor on the right. Figure 4.17b shows the *Metadata & Export* workspace where the exhibition metadata can be changed and the website and Android dynamic feature modules can be exported.

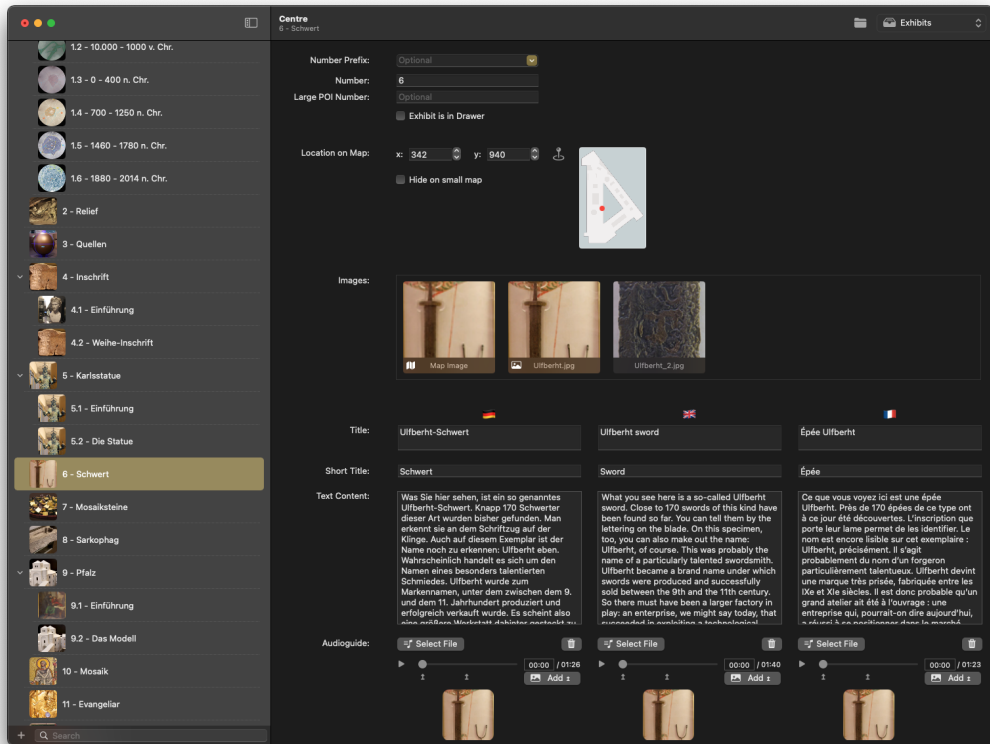


(a) Map & Style workspace in light appearance.

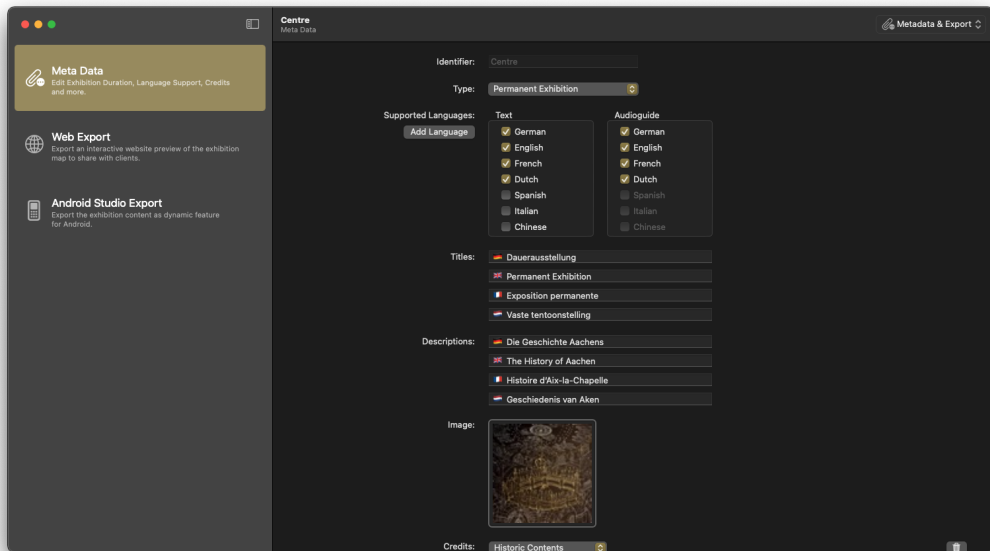


(b) Map & Style workspace in dark appearance with static previews shown on the right.

**Figure 4.16:** The Map & Style workspace is the WYSIWYG part of the content editor. Style attributes are changed in the sidebar on the left and changes are directly displayed within the map. The map allows to position the exhibits via drag and drop. Different languages and appearances can be previewed from controls in the toolbar.



(a) The Exhibit Editor workspace.



(b) The Metadata &amp; Export workspace.

**Figure 4.17:** The workspaces for the Exhibit Editor (a) and Metadata & Export (b). The workspaces can quickly be changed with the popup button on the right side of the toolbar.

## Chapter 5

# User Study

We conducted a user study based on our feature complete software prototype in order to evaluate the usability of the user interface and get a better understanding to what kind of interactions users might expect from a museum content editor.

### 5.1 Design and Execution

The user study consisted of five tasks that tested the most commonly expected user interactions. Due to the pandemic the study was done completely remote via *Zoom* with *Remote Control*. Participants were able to control the hosts mouse and keyboard from their own computer.

Before the participants started interacting with the content editor, we asked them to fill out a questionnaire to evaluate their general level of experience with computers, what applications they typically use and whether or not they had used the operating system before. We also gave them the opportunity to test out the Centre app and explained some of the features of the mobile app.

We conducted a remote user study to test user interactions.

We used silent observation while participants solved the tasks.

We did a semistructured interview to learn more about the participant's reasoning.

Then, we gave the participants the first task along with a folder containing sample files in the same format as they would be in a real-world scenario. We asked them to solve the task without any assistance if possible and used silent observation to analyze the participants actions.

Once the participant believed to have successfully solved the task we followed up with an interview. We used a semistructured interview approach [Lazar et al., 2017] and focussed our questions on any issues observed during the task to find out what elements prompted certain interactions. Furthermore, if participants hadn't discovered certain interactions, we asked them where or how they would have expected it to be instead. Following the interview, the participant would be handed the next task.

Once all five tasks were completed, we asked the participants to fill out a second questionnaire where they would rate four factors of the study: the mental demand during the tasks, their confidence levels to achieve the desired results, how much effort they had to put into it and their frustration levels during the tasks. All four factors are rated from 1 to 7 with 1 indicating a better result. We were, however, more interested in general usability issues and discuss the results of these quantifiable data in Appendix B.

In total, 14 people participated in the user study. Six participants were female and eight male. The participants aged between 18 and 55 ( $\bar{x} = 25.14$ ,  $\sigma = 9.78$ ).

## 5.2 Task 1: Creating Exhibitions

### Task Description

The first task was designed to test the setup process for a new exhibition. Study participants were presented with the initial empty *welcome window* of the content editor. Their task was to create a new exhibition with a given name and a deviating set of supported languages. Furthermore, they were provided with folders containing the audio files.



## Results

All participants had no trouble to find the *Exhibition Setup Assistant* that was designed to guide them through the task. The setup process itself did not pose any problems either. All participants followed the structured approach of the window from top to bottom as intended. Additionally, all participants used the audio folder batch import as well. Albeit four of the participants read the accompanying fine print more closely before deciding to do so. We actually expected this considering that the batch import is more of a convenience pro-feature. For this reason we had added the fine print in the first place and not hidden it away in a tooltip.

Creating new exhibitions posed no problems.

## 5.3 Task 2: Editing Style

### Task Description

The second task tested the interactions necessary to change the styling of the exhibition. We presented the participants with two images depicting the desired look for exhibits on the map for both light and dark appearance. This included changes to all different attributes (colors, fonts, numbers) as well as different styles in light and dark appearance.

## Results

Editing style in different appearances turned out to be the greatest issue in the study. No participant was able to easily find the appearance toggle in the toolbar. Three participants were also unable to find the appearance toggle without help which we considered an unsuccessful completion of the task.

Participants had trouble discovering the appearance toggle.

The main source of confusion was that the map per default displayed the light appearance and whenever the participant would try to edit a style attribute in dark appearance



**Figure 5.1:** A floating bar at the bottom of the map showing the appearance selector on the left and the language selector on the right. This way, it has a stronger association with the map and easier discoverability.

those changes would not be reflected on the map. This led to further confusion about whether or not they were doing something wrong or if they might be editing a wrong style attribute altogether.

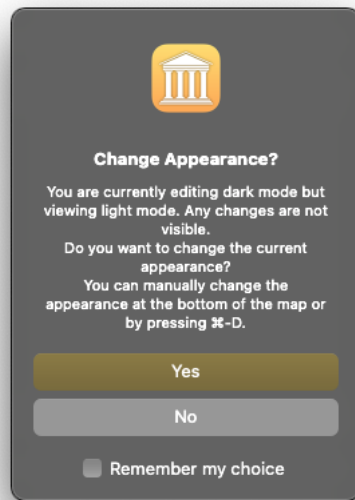
## Discussion

We increased the discoverability of the appearance toggle.

We addressed this issue by changing two things. Firstly, we moved both the appearance toggle together with the language picker from the toolbar to a footer bar on the map. The goal was to have a stronger association with the map as it is floating on top of it as well as moving it to a more unique place to better gain the users eye and thereby increasing its discoverability. This new design is shown in Figure 5.1.

We added an alert to teach the user about different appearances.

Secondly, we added an additional dialog that shows up after the user has edited a color in a different appearance than his current one (Figure 5.2). This dialog informs the user about why he does not see his recent changes reflected, explains where to change it and offers to change the appearance for the user. Furthermore, it offers a *Remember my choice*-checkbox, basically allowing the user to automatically switch the appearance if they choose to. This setting can be changed at any time in the *Preferences* window of the content editor.



**Figure 5.2:** An info dialog informing the user about different appearances.

## 5.4 Task 3: Editing Exhibits

### Task Description

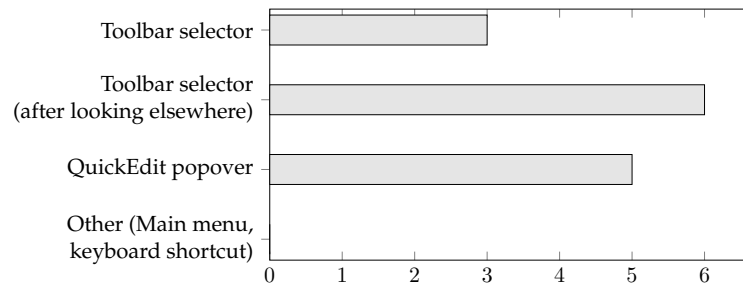
The third task tested the editing of exhibits. Participants had to position the exhibits on the map based on a sketch and fill out one exhibit based on images and text files provided to the participant.

### Results

All participants arranged the exhibits on the map using drag and drop.

To fill out all the other information, the participants had to change the workspace from *Map* to *Editor*. The different strategies are depicted in Figure 5.3. While every participant was able to achieve it, only three users found the

Participants had different notions of where to change the workspace.



**Figure 5.3:** Comparison of how study participants changed the workspace. About one third did not discover the workspace selector in the toolbar.

workspace selector in the toolbar instantly. Six participants found it in the toolbar after looking in other places first and five participants used the QuickEdit popover to change the workspace. Two participants were uncertain if changing the workspace without saving would result in loss of data.

The automatically generated map thumbnail image was confusing.

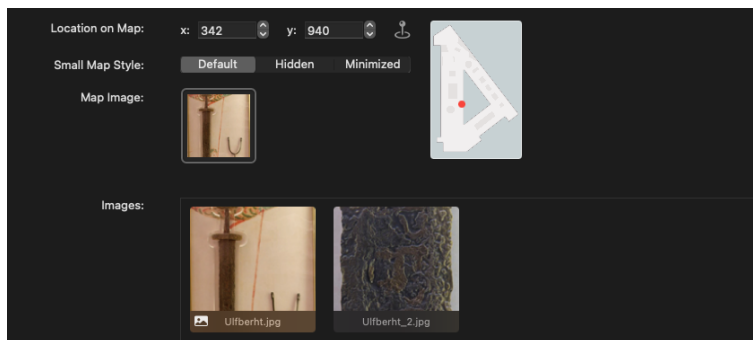
Another issue was that the content editor automatically creates a map thumbnail image when the first image is added. It confused all participants that instead of the one image that they added two appeared instead. Though they were able to figure out what happened based on the map icon underneath the image.

We also observed how the participants entered the localized content into the editor. Nine participants filled out a single value in each language. Five participants filled out a single language at the time.

## Discussion

There is no standardized position for a workspace selector.

Since the discoverability of multiple workspaces had been one of our concerns during the design phase, we followed up on this during the interview. Those participants who had less problems changing the workspaces had previously worked with software that also uses multiple workspaces. Those included *Affinity Photo*, *Adobe Photoshop*, *DaVinci Resolve*, and *Blender*. However, to the best of our knowledge, there is no standard of where to place the workspace se-



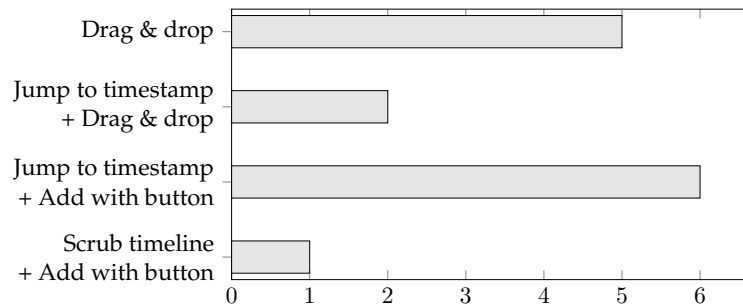
**Figure 5.4:** To avoid confusion the map thumbnail image is now in its own separate row.

lector. Some apps use the top right corner of the toolbar (where we had placed it based on our own experiences), others the top left corner of the toolbar and others use a bottom tab bar. Study participants would start to look for a workspace selector based on their previous experiences.

Without a standardized position, we reconsidered our initial placement of the workspace selector and thought about where it would best be placed logically and best for usability and discoverability. The left side of the toolbar is traditionally intended for navigational items on macOS (e.g., the back and forwards button in Safari). Marking a toolbar item as navigational even places them before the window title by default. The workspace selector can be considered navigational and its more prominent position before the window title helps with discoverability. This also makes sense from a usability point of view: When the user changes the workspace from *Map* to *Editor* the most probable next action is to select an exhibit in the sidebar. If the workspace selector is placed on the left, above the sidebar, mouse travel distance is reduced which makes changing the workspace faster (Fitts's Law).

We changed the position of the workspace selector to the left side of the toolbar for better discoverability.

To avoid the confusion regarding the map thumbnail image, we decided to remove it from the collection view of all images and instead add a separate row for the map image (see Figure 5.4). That way it is also grouped together with the other map related values.



**Figure 5.5:** Comparison of different kinds of user interaction to add keyframes to the audio player. Half the users used drag and drop, the other half added them with a button.

## 5.5 Task 4: Audio Player Interactions

### Task Description

In the fourth task we tested interactions with the audio player. Participants first had to place three images at certain timestamps. Then they would be asked to change a timestamp afterwards.

### Results

Participants chose different ways to place keyframes.

The results of the initial placement are shown in Figure [5.5](#). Five participants placed the images with drag and drop directly onto the timeline. Two used the current time input text field to jump to the required timestamp and then used drag and drop. Both participants used drag and drop without the time input after the first image though. Six participants used the time input text field to jump to the timestamp and use the *Add Keyframe* button. And one participant moved the timeline slider to the correct timestamp and used the button then.

When asked to change one keyframe, four participants moved it with drag and drop. Three participants simply deleted the keyframe via the context menu and added it

again at the new timestamp and seven participants used the popover to directly set the new timestamp.

## Discussion

When we designed the audio player UI and interactions, we deliberately added multiple ways to add keyframes because we were not sure what kind of interaction users would expect. The results of this study show that in this case, it was an even 50:50 split between adding keyframes via drag and drop and first finding the correct timestamp before adding the keyframe via a button.

Likewise, participants changed keyframe positions using all the different interactions that we had added (plus deleting the keyframe and adding it again). We had already experienced this during our test with the app development team and the user study showed similar results. We believe that this might be due to keyframes not being very commonly used in other applications apart from animation or video editing software. Most study participants had no prior experiences with keyframes and first had to learn how to interact with them. So it is probably good to have multiple options to better accommodate users unfamiliar with the keyframes.

Participants had little experience with keyframes.

## 5.6 Task 5: New Exhibits and Children

### Task Description

In the fifth and final task, participants had to create a new exhibit with a given location on the map. Then they had to create a new group with the new exhibit and an existing one (to create child exhibits).

## Results

Ten participants created the new exhibit in the *Editor* workspace from the sidebar and would then change the location on the map. Four participants used the *Map* workspace to create the exhibit right at the desired location.

Some participants tried to group exhibits on the map.

To create a group, three participants used drag and drop in the sidebar. Six participants selected both exhibits in the sidebar and created a group via the context menu. Four participants tried to group the exhibits on the map by either attempting to drop them on top of each other (3) or by drawing a selection frame in an attempt to select both exhibits (1). When this wasn't successful, all four changed to the *Editor* workspace and grouped them via the context menu. One participant was unable to group the exhibits at all which we considered an unsuccessful completion of the task.

## Discussion

The attempts to group exhibits on the map was something that we hadn't considered prior to the study. Our own mental model [Norman, 2013] was that managing exhibits would take place in the *Editor* workspace and that the *Map* workspace was used as a live preview along with the ability to edit style.

We added the ability to group exhibits on the map.

And although only four participants attempted to group exhibits on the map, their intentions made sense: An exhibit group is required if two exhibits are too close to each other or even overlap. This is something that is only noticeable on the map and just then a user would decide to group them. Therefore, we added the option to create exhibit groups from the map.






**Figure 5.6:** The map allows the user to select multiple exhibits at once.

## 5.7 Conclusions

The intent of the user study was to find out about general usability issues as well as to learn what kind of interactions users expected for certain tasks. While we were more interested in general trends and issues, we evaluated individual user interactions and feedback as well. Two of them we want to discuss further.

Firstly, one study participant tried to draw a selection frame around multiple exhibits on the map (see Chapter 5.6). Although this was a single occurrence, selecting multiple exhibits on the map certainly has benefits. It is not just useful for grouping exhibits (as intended by the participant) but also allows the user to move multiple exhibits at the same time. We added the option to select multiple exhibits on the map either by drawing a selection frame or by clicking while pressing . However, to not mess up the WYSIWYG aspect of the map, we couldn't highlight selected exhibits by, for instance, a bordered frame around them or an opaque overlay. We decided that, when multiple exhibits are selected, they are displayed just as they are supposed to be whereas all exhibits currently not selected have a slightly reduced opacity as shown in Figure 5.6.

We added multiple selection of exhibits to the map.

We made the app more consistent with macOS.

The second change was based on feedback by a single participant: During the interview after completing task 2, they pointed out that it would make sense to move the style sidebar from the left side of the window to the right. That way it would follow the established macOS window layout: navigation on the left, content in the middle and tools or inspector on the right. Though we couldn't find a guideline explicitly stating this, it is used throughout the system (e.g., in Finder, iWork, Xcode, Photos). Our placement of the style on the left came from the initial three-pane concept and we hadn't revisited it ever since. But placing it on the right side of the window made sense and better matches users expectations.

In conclusion, the user study confirmed that the content editor is mostly working as intended. We addressed the one major usability issue (editing style attributes in the currently inactive appearance; see Chapter [5.3](#)) and adjusted some UI elements based on user feedback.

The use of multiple workspaces, once discovered, worked really well. Although all participants were inclined to look for a solution at their current workspace first. If that was unsuccessful, they explored other workspaces to check if the task could be solved there.

## Chapter 6

# Final Product and Conclusions

With the user study completed, we added some finishing touches to the content editor and prepared it for its first real-world usage.

### 6.1 Finishing Touches

The final changes to the content editor were mostly quality-of-life improvements but did also contribute to the overall usability. Firstly, we analyzed where users might experience slight delays and would require some sort of feedback (Nielsen's first usability heuristic). A small (~one second) delay happens when an exhibition is opened from the *welcome window*<sup>1</sup>. We added progress indicators to inform the user that the exhibition is indeed loading. We also use progress indicators when creating a new exhibition and using the exhibit batch import.

We added more feedback for the user.

We also improved the validation of menu items. Menu items that only work on a certain workspace are hidden for

We improved menu validation.

---

<sup>1</sup>The delay varies based the size of the exhibition and the speed of the data storage. The one second delay was measured opening the permanent exhibition (61 exhibits) from an SSD.

all other workspaces. And menu items that can't be performed (e.g., because they require at least two exhibits to be selected and currently only one is selected) are grayed out to indicate to the user that their action is not possible at the moment.

Finally, we improved the workflow for adding exhibits directly on the map. Previously, that would simply create a new empty exhibit at the position of the click. Now the *QuickEdit* popover opens automatically so that the user can name the new exhibit right away.

With these changes, the content editor was ready for its first use in a real-world scenario.

## 6.2 Real-World Application and New Features

The content editor has successfully been used by the app development team during their work for an upcoming exhibition. Their feedback was that working with the content editor is quite nice and a lot faster than before.

An upcoming exhibition needs additional features.

The upcoming exhibition has a new feature concerning the style. Unlike previous exhibitions, where some exhibits would simply be hidden on the scrolled out map (small map), the new exhibition would instead decrease the size of exhibits. So on a small map exhibits can either be displayed normal (i.e., displayed according to the small style attributes), hidden, or minimized. The new minimized style would also use a different background color. We have previously named expandability as a requirement and with this new style we had an opportunity to test this claim.

Adding the new style to the exhibit was quite simple. Instead of a `Bool` we used an `enum`. New exhibits use the `enum` automatically and opening an old exhibit without the new style attribute in its JSON simply falls back onto the old `Bool` value. In the user interface, we only needed to swap the checkbox to a segmented control and add the new

minimized case to the `ExhibitView` that is displayed on the map.

Adding the new background color to the style was just as simple and only required adding it in four places in code: The `Codable` struct that reads from the “`Style.json`” file, the `Style` class where the new value is properly cast and saved, in the style `NSCollectionViewItem` where the new value is added to the user interface along with a getter and setter, and in the `ExhibitView` where the new background color is applied. Due to our use of `Combine` these are the only necessary changes: two to add it to the model and two to add it to the user interface.

Only four edits are necessary to add new style attributes.

## 6.3 Conclusions

Our content editor was designed as an all-in-one solution for creating and maintaining exhibitions for the Centre app. The resulting app has numerous advantages over manually editing the files. Most noteworthy are the improved usability, faster completion of tasks (due to WYSIWYG and automation), and a correct (no syntax errors) generation of files. One disadvantage is that the content editor is one more software to maintain. We try to mitigate this by keeping the code close to the iOS app, but adding a new feature to the mobile apps still requires additional steps to add them to the content editor as well.

We now go over three aspects of our work that had the largest impacts during the development:

The first one is the separation of different workflows into individual workspaces. With museum content being as extensive and diverse as it is and many different types of workflows, it was the right decision to use workspaces. Not only does it allow the user to focus on a single task at a time, but all other workflows are still possible in the context of a single window.

Different workspaces help to keep the interface clean and users focused.

WYSIWYG speeds up former workflows tremendously.

Secondly, the WYSIWYG part of the content editor probably expedites the former workflows the most. The content editor reduces the time to preview changes from ~12 seconds to instantaneous. It also retains the context for the user. They no longer have to switch between programs (from a text editor to Xcode to the simulator) for a single task.

Reactive programming helps to keep the code cleaner.

Lastly, by using reactive programming we were able to write easily expandable code. At the same time we were able to reduce the amount of code necessary in the controller; one of the issues with traditional MVC [Dobrea and Diosan, 2019].

Finally, we want to discuss the Return on Investment (ROI). Is it worth the time (and money) to build a content editor for museum guides? It certainly takes some time to develop a content editor. But using the content editor saves a lot of time in the long run. Developing a content editor makes sense if a museum has regularly changing temporary exhibitions or if exhibits are routinely changed. The development time for a content editor can be further decreased if the model or other code of an already existing mobile app can be reused in the content editor.

## Chapter 7

# Summary and Future Work

In the following chapter, we conclude the thesis with a summary of our findings and give an outlook into possible future work.

### 7.1 Summary and Contributions

In this thesis, we described the development and implementation of a content editor for museum guides. In Chapter 2 “[Related Work](#)”, we had a look at research regarding WYSIWYG content editors in general and discussed their advantages and disadvantages. We also discussed research regarding museum guides. This included research into context-aware information and augmented reality. Lastly, we had a look at the Centre mobile app that our content editor would be based on.

In Chapter 3 “[Initial Considerations](#)”, we discussed that there are two stakeholders involved in the project (the people at the museum and the app development team) and that exchanging content between them is one of the challenges. Other challenges and limitations are a complex, rigid data structure for the app and a tedious and lengthy process to

preview changes. Based on this we presented a list of requirements for the content editor.

In Chapter 4 “Iterative User Interface Design”, we described the design process of the content editor in increasing fidelity starting with paper prototypes of the whole user interface. We presented some digital mock-ups before we described a horizontal software prototype. Here, we had a look at some individual features of the content editor, how they evolved, and how they contributed to an overall increased usability. Finally, we presented a feature complete software prototype.

We evaluated the usability of our content editor in Chapter 5 “User Study”, where we discovered one major usability issue when editing style values in different appearances. We addressed this and also improved the discoverability of the different workspaces. Apart from this, we learned that the editor was indeed working as intended.

In Chapter 6 “Final Product and Conclusions”, we concluded our iterative design process with some finishing touches. We prepared the content editor for its first real-world application and tested its expandability by adding a new feature. Finally, we presented our conclusions for the development process as whole.

## 7.2 Future Work

We implemented our museum content editor with all the features that are currently needed for the Centre app. However, any future work on the Centre app can also extend to the content editor.

One near-term addition could be beacon technology to localize the visitors inside the museum. This is a suitable addition to the WYSIWYG part of the app. The content editor can show the beacons on the map and possibly even help with triangulation. We have already included marks in the code where additional code for beacons can be added.



Another possibility for future work is AR content. As we have already outlined in Chapter 2.2, museums can use AR content in many different ways. Adding AR content to the content editor would probably be a completely different workflow and also need more space. Therefore, it would be another example of an additional workspace. For instance, the content editor could use [SceneKit](https://developer.apple.com/scenekit/)<sup>1</sup> to show previews of 3D objects used in AR.

---

<sup>1</sup><https://developer.apple.com/scenekit/>



## Appendix A

# Full List of Requirements

We created the following list of requirements by observing the workflows of the app development team. It includes additional, initial feature requests and wishes. It is not, however, a list of all features in the final version of the content editor. During the design process, we discovered the need for additional features which we discuss in Chapter [4](#). All these initial requirements are implemented in the final product.

- Create new exhibitions
  - Create required folders and files
  - Create metadata (languages, credits)
  - Batch import existing audio folders with an exhibit for each audio file
- Style editor
  - WYSIWYG style preview
  - Pipette tool
  - Only save deviation from default style
  - Reset to default

- Live map
  - Map with exhibits correctly positioned
  - Content preview for audio guide and text content
- Exhibit editor
  - Tree structure sorted by id
  - Warning / error indicators for missing values
  - Convert text content into HTML files (maintain bold, italic, underline)
  - Import images, resize to fixed (settings adjustable) size
  - Play audio files and set keyframes
- Export / Import
  - Export website for the museum
    - \* Work without server, without browser security adjustments
    - \* Show map with exhibits, supports moving via drag and drop
    - \* Available in German
    - \* Send changes back via email as JSON string
  - Import the website export JSON string
  - Export content as dynamic feature module for Android

## Appendix B

# Quantifiable User Study Results

Here, we discuss the results of the user study questionnaires and other quantifiable data.

All participants were told that their goal is to solve the tasks correctly rather than fast. But we did measure how long it took to complete all tasks (including filling out the questionnaire). The required time was between 0:36 and 1:44 ( $\bar{x} = 1 : 05$ ,  $\sigma = 0 : 16$ ). If we exclude the two participants who did not have any prior experience with macOS at all, we get 0:36 to 1:15 ( $\bar{x} = 0 : 59$ ,  $\sigma = 0 : 11$ ).

After participants had successfully completed all five tasks, they were asked to rate four factors. These questions were inspired by the [NASA Task Load Index](#)<sup>1</sup>, but tailored to our study and with a range from 1 to 7. The results inside the brackets are the average if we exclude the two participants without macOS experience.

- **Mental Demand:** This asked the participants how mentally demanding the tasks had been with 1 being *not demanding* and 7 being *very demanding*. Participants rated it  $\bar{x} = 2.357$  ( $\bar{x} = 2.167$ ).

---

<sup>1</sup><https://humansystems.arc.nasa.gov/groups/TLX/>

- **Confidence:** This asked the participants how confident they were to achieve the desired results with 1 being *very confident* and 7 being *little confident*. Participants rated it  $\bar{x} = 3.071$  ( $\bar{x} = 2.5$ ).
- **Effort:** This asked participants how hard they had to work to accomplish the results with 1 meaning *little effort* and 7 *much effort* required. Participants rated it  $\bar{x} = 2.714$  ( $\bar{x} = 2.25$ ).
- **Frustration:** This asked participants how frustrated and stressed they felt while performing the tasks with 1 meaning *little frustration* and 7 meaning *much frustration*. Participants rated it  $\bar{x} = 2.643$  ( $\bar{x} = 2.083$ ).

While all four factors are reasonably low, the worst rated factor is the confidence level indicating whether or not participants believed that their actions would yield the desired results. We think that there are two main reasons for this: Firstly, the content editor offers a lot of functionality. Although we tried to keep the user interface as clean as possible, there are still a lot of UI elements that the participants can interact with. Secondly, participants were unfamiliar with the features of the Centre app prior to the study. We gave them the opportunity to test out the app before we began the study. However, by briefly using the app we can't expect study participants to have the same knowledge about all the different features that a member of the app development team would have.

## Bibliography

Bashar Al Takrouri, Karen Detken, Carlos Martinez, Mari Klara Oja, Steve Stein, Luo Zhu, and Andreas Schrader. Mobile holstentour: Contextualized multimedia museum guide. In *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pages 460–463, 2008.

Eun Sok Bae, Dong Uk Im, and Sung Young Lee. Smart museum based on regional unified app. *International Journal of Software Engineering and Its Applications*, 7(4):157–166, 2013.

Raluca Budiu. Interaction cost. <https://www.nngroup.com/articles/interaction-cost-definition/>, 2013. [Online; accessed 10-September-2021].

Raluca Budiu. Wizards: Definition and design recommendations. <https://www.nngroup.com/articles/wizards/>, 2017. [Online; accessed 10-September-2021].

Donald D Chamberlin. Document convergence in an interactive formatting system. *IBM journal of research and development*, 31(1):58–72, 1987.

Paul Chandler and John Sweller. Cognitive load theory and the format of instruction. *Cognition and instruction*, 8(4): 293–332, 1991.

John J Chester and Arturo J Sánchez-Ruíz. Building a web-based, WYSIWYG interface cascading style sheet editor.

Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. Experiences of developing and deploying a

- context-aware tourist guide: The GUIDE Project. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 20–31, 2000.
- Mandy Ding et al. Augmented reality in museums. *Museums & augmented reality—A collection of essays from the arts management and technology laboratory*, pages 1–15, 2017.
- Dragos Dobrean and Laura Diosan. Model view controller in ios mobile applications development. In *SEKE*, pages 547–716, 2019.
- Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.
- Helen Jenny, Andreas Neumann, Bernhard Jenny, and Lorenz Hurni. A WYSIWYG interface for user-friendly access to geospatial data collections. In *Preservation in digital cartography*, pages 221–238. Springer, 2010.
- Joel Lanir, Tsvi Kuflik, Alan J Wecker, Oliviero Stock, and Massimo Zancanaro. Examining proactiveness and choice in a location-aware mobile museum guide. *Interacting with Computers*, 23(5):513–524, 2011.
- Soren Lauesen and Morten Borup Harning. Virtual windows: Linking user tasks, data models, and interface design. *IEEE software*, 18(4):67, 2001.
- Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research Methods in Human-Computer Interaction*. Morgan Kaufmann, 2017.
- Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.
- Don Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic books, 2013.
- Carmen Santoro, Fabio Paterno, Giulia Ricci, and Barbara Leporini. A multimodal mobile museum guide for all. *Mobile Interaction with the Real World (MIRW 2007)*, pages 21–25, 2007.



- Ben Shneiderman, Catherine Plaisant, Maxine S Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, 2016.
- Jacqueline Spiesser and Les Kitchen. Optimization of HTML automatically generated by WYSIWYG programs. In *Proceedings of the 13th international conference on World Wide Web*, pages 355–364, 2004.
- Vassilios Vlahakis, Nikolaos Ioannidis, John Karigiannis, Manolis Tsotros, Michael Gounaris, Didier Stricker, Tim Gleue, Patrick Daehne, and Luis Almeida. Archeoguide: An augmented reality guide for archaeological sites. *IEEE Computer Graphics and Applications*, 22(5):52–60, 2002.
- Leonard Wein. Visual recognition in museum guide apps: Do visitors want it? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 635–638, 2014.
- Max Wertheimer. Laws of organization in perceptual forms. 1938.
- Nick Williams and Tim Wilkinson. Experiences in writing a WYSIWYG editor for HTML. In *Proceedings of WWW*, volume 94. Citeseer, 1994.
- David Wolber, Yingfeng Su, and Yih Tsung Chiang. Designing dynamic web pages and persistence in the WYSIWYG interface. In *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 228–229, 2002.
- Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. WYSIWYG development of data driven web applications. *Proceedings of the VLDB Endowment*, 1(1): 163–175, 2008.
- Faheem Zafari, Athanasios Gkelias, and Kin K Leung. A survey of indoor localization systems and technologies. *IEEE Communications Surveys & Tutorials*, 21(3): 2568–2599, 2019.



---

# Index

Combine .....	17	26-27	54-55
Keyframe .....	30-32	48-49	
Outline View .....	22-23	33-34	
Prototypes			
- Paper .....	20	21	
- Digital Mock-up .....	22	23	
- Software .....	39	40	
Requirements .....	15-16	61-62	
Website .....	16	18	35-36
Workspace .....	23-24	45-47	

