Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

Aufbau eines Frameworks für digitale diagnostische Interviews

Building a framework for digital diagnostic interviews

Masterarbeit
Informatik

2023

Vorgelegt von Presented by	Felix Nickels Matrikelnummer: 421701 felix.nickels@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Dr. Ruth von Brachel Lehr- und Forschungsgebiet: Klinische Kinder- und Jugendpsychologie Ruhr-University Bochum
Betreuer Supervisor	Dr. rer. nat. Pascal Richter Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Contents

1 Introduction

Technological advancements have transformed various industries in today's digital era, and clinical psychology is no exception. With the increasing prevalence of mental disorders - particularly due to the COVID-19 pandemic [18] - and the need for suitable treatments, it is crucial to leverage technology to improve the efficiency and accessibility of therapeutic practices. This thesis focuses on developing an app specifically for psychotherapists, aiming to replace the conventional paper-based interview guideline currently utilized by psychotherapists.

The prevalence of mental disorder continues to exhibit an upward trend, highlighting the pressing need for effective treatments and diagnostic tools.[1] Structured interviews are well known for an accurate diagnostic assessment, creating the basis for suitable therapies. Furthermore, they are the gold standard in the diagnostic process, offering a comprehensive understanding of the patient's symptomatology. Despite the recognized benefits of structured interviews, they are rarely used because of concerns about their acceptance and feasibility. One explanation might be psychotherapists' worries that structured interviews would be ineffective or harm the therapeutic relationship [11]. However, previous studies with adults, children, adolescents and their parents on accepting structured interviews do not support these considerations [15]. Another reason for the low usage is that psychotherapists prefer to rely on their judgment, although it would lead to unreliable diagnoses. Another argument for not using structured interviews is about time since they are perceived as time-consuming, requiring additional time and resources that psychotherapists may find impractical. Furthermore, psychotherapists may feel uncomfortable using structured interviews and perceive them as rigid and inflexible. [16, 3]

The primary objective of this research is to develop an app that addresses the challenges previously mentioned, providing a user-friendly interface for conducting structured interviews with ease. By digitizing the existing paper-based guideline, the app aims to overcome the difficulties and barriers to using structured interviews and providing accurate diagnoses.

1.1 Related Work

At the time of this thesis, only three digital applications exist in addition to the paper-based interview guideline. In the following, the interview guideline that is currently mainly used and the three digital applications are described.

[1] https://www.who.int/news-room/fact-sheets/detail/mental-disorders; https://www.statista.com/statistics/979869/number-of-people-with-mental-health-disorders-globally/

1.1.1 Paper-based Interview Guideline

Currently, there is no structured interview method without using the paper-based interview guideline. This guideline is based on the DIPS (Diagnostische Interviews für Psychische Störungen). It is used to classify mental disorders. The interviewer asks precisely defined questions to his client. After the interview, an evaluation is made to determine the fulfilled disorders based on the client's answers. [14]

1.1.2 DGPPN

The mobile app DGPPN (Deutsche Gesellschaft für Psychiatrie und Psychotherapie, Psychosomatik und Nervenheilkunde). It is available for both Android and iOS platforms. The app provides a comprehensive platform for accessing guidelines, publications, and resources in the mental health field. It offers up-to-date information on psychiatry, psychotherapy, and psychosomatic medicine. These guidelines serve as evidence-based recommendations for diagnosis and treatment.[2]

1.1.3 Online Survey

It exists a digital version of a similar structured interview that is used with children from the age of six. This online tool is a copy of the paper version embedded in an online survey tool based on the platform REDCap[3]. It does not contain diagnostic suggestions, nor does it adapt the interview content to the children's age or the given answers.[4]

1.1.4 testbox

It is an online assistant for standardized and test-based diagnostics. The user can choose from a collection of different test types. Next, the client performs this test at any place on any device. Afterwards, the test is automatically evaluated, and the user receives the results on his dashboard. This result is used in combination with a scale to determine how strong a disorder is. testbox does not offer diagnostic interviews but disorder-specific tests. As a result, it cannot create or reject a client's diagnosis but determines the strength of a client's already known diagnosis using these tests.[5]

1.2 Contribution

This thesis will focus on designing, developing, and evaluating the app, specifically targeting its usability, effectiveness, and impact on the diagnostic process. The research will involve cooperating with psychotherapists at the Ruhr University Bochum (RUB)

[2]https://www.dgppn.de/leitlinien-publikationen/die-dgppn-app.html
[3]https://www.project-redcap.org
[4]https://redcap.zih.tu-dresden.de/redcap/surveys/?s=A9497MKE7MDXFR7D
[5]https://testbox.de

of the chair of Clinical Child and Adolescent Psychology at the Forschungs- und Behandlungszentrum für psychische Gesundheit (FBZ)[6] to understand their perspectives, preferences, and concerns regarding the use of structured interviews and technology in their practice, research and teaching. The app will be designed to be compatible with both Android and iOS platforms, ensuring accessibility and convenience for psychotherapists.

1.3 Outline

The second section of the thesis focuses on explaining the tools used in the development process. This includes a detailed description and justification of the specific tools and technologies. The UI/UX principles used to develop the app are covered in section 2.2. It explains how these concepts were used to produce a clear and user-friendly app interface, going into the fundamentals and recommended user interface and user experience design methods. Furthermore, section 2.3 describes the chosen development approach. The existing procedure, particularly the functioning of the paper-based interview guideline, is explained in detail in section 3 to give an in-depth understanding of the app's functionality. Next, the data model is presented, explaining the underlying data structure used in the app. It discusses the design and organization of the data model, focusing on how it supports the app's functionalities and facilitates data storage and retrieval. The major software components are then explained by discussing their functionalities and interdependencies. After all technical elements are explained, a demonstration of the app's features provides an overview of its user interface and functionality in section 4. Through a comparison that incorporates a user study, the performance and usability of the app are critically assessed. Section 5 compares the app with the conventional paper-based interview guideline while presenting the user study's findings and analysis. Finally, the thesis concludes with a comprehensive summary of the findings, drawing conclusions based on the comparison and user feedback. It also discusses the implications of the findings and suggests areas for further development.

2 Tools and Principles

This section introduces the toolbox by clarifying the tools. The applied design principles are explained to follow the design choices made to provide an engaging design that offers the best user experience and usability. This is followed by an explanation of the user-centered design approach.

2.1 Tools

First, the tools used are described, beginning with the prototyping tools and moving on to the development tools.

[6] https://www.kli.psy.ruhr-uni-bochum.de/fbz/fbz.html

2.1.1 Figma

Figma is a powerful web-based tool for collaborative interface design. Additionally, Figma provides mobile apps for both Android and iOS, enabling designers to work seamlessly across various devices. Its wide range of features combines multiple vector graphics editors and prototype tools with a specific emphasis on user interface (UI) and user experience (UX) design.
The primary purpose of Figma is to enable the creation of medium to high-fidelity prototypes. Designers can easily refine their prototypes with each iteration fostering an iterative design process. This is due to the building of reusable components of UI elements. In addition, the designer can add interaction sequences to UI elements to encourage the user to interact with the prototypes. Moreover, Figma offers a file format known as FigJam, which serves as an online whiteboard tool capable of creating diverse visual diagrams such as mind maps. By utilizing FigJam files, designers can, for example, effectively map workflows into flow charts, representing the data flow within the UI.
Since Figma can work collaboratively, it can be used simply by multiple stakeholders to work together in real-time. Designers can share their work easily, obtain feedback and make changes.[7]

2.1.2 Flutter

The decision to use Flutter for this project arose from the requirement to develop an application compatible with both Android and iOS platforms. Flutter, developed by Google, proved an optimal choice due to its ability to create cross-platform applications using one single codebase. Besides Android and iOS, Flutter's single codebase can be used for web applications, desktop applications (macOS, Windows, Linux), and embedded systems.
In Flutter, widgets are the building blocks of the UI, representing different elements such as buttons, text, images, and more. They are divided into stateful and stateless widgets playing an essential role in developing efficient and responsive Flutter applications. As the name implies, a stateful widget manages and maintains its internal state. A stateful widget can change its behaviour or appearance whenever something changes, such as data or other events. When UI elements must be dynamically modified due to user interactions, data changes, or other events, the stateful widget is rebuilt to reflect its new state. They provide a powerful mechanism for managing and reacting to changes in state. On the other hand, stateless widgets are immutable and do not have any internal state. They are used to presenting static content. Stateless widgets are based on the data they provided at the creation time. Since they do not need to manage an internal state, they are lightweight and efficient reducing unnecessary rebuilds and minimizing memory consumption.[8]
Flutter uses the Dart programming language as its primary language for application

[7] https://www.figma.com
[8] https://flutter.dev

development. Dart is an object-oriented and class-based language that incorporates garbage collection to efficiently manage memory. With its syntax similar to Java and JavaScript, Dart offers a familiar environment for developers transitioning from other programming paradigms. The code written in Dart is compiled into machine code (or JavaScript in the case of web applications), optimizing the performance and ensuring native-like execution on Android and iOS devices.[9]

2.1.3 Visual Studio Code

Visual Studio Code is a free and open-source code editor developed by Microsoft that is available for Windows, Linux, and macOS. It provides debugging, syntax highlighting, code completion, and refactoring features. The editor also includes IntelliSense, a feature that provides smart completions based on variable types, function definitions, and imported modules, which goes beyond syntax highlighting and autocomplete. An outstanding aspect of Visual Studio Code is its rich extension ecosystem that adds functionality, such as support for additional programming languages, themes, debuggers, commands, and more. These extensions are developed by both Microsoft and the community.[10] In this project, the extensions Dart and Flutter are used.[11]

2.1.4 Device Simulators

Testing on respective devices is essential to ensure the compatibility and functionality of the app across both Android and iOS. Specialized integrated development environments (IDEs) such as Android Studio and Xcode are used to facilitate the process of testing.
Android Studio, the official IDE for Android app development, provides a comprehensive set of tools, including the Device Manager. Virtual Android devices (AVD) can be created in the Device Manager, emulating the behaviour and characteristics of a real Android device. An AVD allows developers to run and test their applications in a controlled environment. This makes identifying and fixing platform-specific issues easier and leads to optimal performance and user experience on Android devices.[12]
Similarly, for iOS app development, Apple offers Xcode as the official IDE. It provides a range of features and functionalities to help developers in creating robust and visually appealing applications for Apple devices. To test the iOS app, developers use Xcode's integrated simulator for various Apple devices such as iPhones and iPads. Using different devices and screen sizes ensures that the app delivers a consistent and optimized experience across the Apple device ecosystem.[13]

[9] https://dart.dev
[10] https://code.visualstudio.com
[11] https://dartcode.org
[12] https://developer.android.com/studio
[13] https://developer.apple.com/xcode

2.1.5 SQLite

In this project, the relational database management system SQLite is chosen to benefit from many advantages.
First, relational databases enforce strict data integrity rules to ensure that data is consistent and accurate. This means that data is organized and stored in a way that avoids duplicates and maintains consistency between tables. Relational databases are scalable and can handle large amounts of data. They can process complex queries and large amounts of data and are suitable for small and large applications. SQL provides a flexible and powerful query language that allows users to extract and manipulate data in a variety of ways. SQL is standardized, widely used, and well-supported by many database tools and frameworks. [10]
Second, SQLite does not need any configuration or setup because it is self-contained and does not require any separate server process. It can directly read and write database files on the hard disk without requiring additional service processes. Thus, the data protection regulation rules can be respected since every data is only located locally on the device. In addition, it is fast, efficient, scalable, and independent of the operating system, such that it can be used in various operating systems. [2]
Furthermore, SQLite supports the ACID principle. It is an acronym for Atomicity, Consistency, Isolation, and Durability. These four key properties guarantee the reliability of transactions in the database. Atomicity means that a transaction is treated as a single, inseparable unit of work. If any part of the transaction fails, the entire transaction is rolled back to ensure the consistency of the database. Consistency refers to the fact that a transaction transfers the database from one consistent state to another consistent state and, at the same time, ensures data integrity. Isolation describes that several transactions can take place simultaneously without interfering with each other. Each transaction is executed in isolation so that the results of one transaction do not affect the results of another transaction. Moreover, Durability implies that once a transaction is committed, its changes are permanent and will survive subsequent failures or system crashes. The database ensures that all changes are written to disk before acknowledging that the transaction is complete. [10]

2.2 User Interface and User Experience Design Principles

In order to comprehend the decisions made regarding the user interface and user experience design, detailed information about diverse rules and patterns is explained.

2.2.1 Fitts' Law

Fitts' Law is a popular law that calculates the necessary movement time for pointing devices. The movement time is calculated as follows:

$$MT = a + b * log_2(\frac{2A}{W})$$

Here a and b stand for regression coefficients and depend on the device. For example, using a finger as a pointing device leads to $a = 100ms$ and $b = 50ms/bit$. The width (W) and the distance (A) to the target are also necessary to determine the movement time. Overall, applying Fitts' Law helps determine the correct width of elements and an acceptable distance between starting point and the target. [13]

2.2.2 Gestalt Laws

Gestalt theory is a psychological theory explaining how human perception works and why humans tend to group things together. Only the most commonly used Gestalt laws in the app are explained since there exists over 100 of them.
The first Gestalt law is about proximity. Objects that are close together are perceived as a group. Imagine a letter. A letter usually has a sender and a recipient. The data of the recipient and the sender are written on the letter, and without further labels or lines, it is clear which data belong to which person. This is because the sender's name, street, and place are written close to each other. The same applies to the recipient. Due to the proximity of the individual lines, this unconsciously creates a grouping of the data of the respective person. In addition, humans have an innate ability to close gaps in shapes, especially when those shapes are known. This means incomplete shapes are perceived as complete without consciously thinking about them. This is called the law of closure. The next law is about similarity. Elements with similar properties (e.g., shape, colour, proximity, direction, size) are perceived as groups. This is also true if the elements have a spatial distance from each other. [6] Lastly, the Gestalt law of common region suggests that objects enclosed within the same region are perceived as a group, regardless of their similarity or proximity [12].

2.2.3 Affordances and Signifiers

Affordances describe the relationship between an object and a person. Both the object's properties and the person's capabilities are important and together determine the affordances. Norman, the author of the book "The Design of Everyday Things" [17], uses a chair to demonstrate that while it would only afford sitting for weak people, it would afford to lift strong ones. Some affordances are not perceived for a variety of reasons. For this, Norman introduces signifiers. Signifiers act as labels and make it possible to perceive affordances. Overall, affordances communicate what actions are possible, and signifiers communicate where the action should occur. [17]

2.2.4 Mappings

The term mapping is derived from mathematics and describes the relationship between the elements of two sets. For example, a set of light switches controls a set of lights in a room. To control the lights without help or explanations, Norman recommends using the concept of natural mappings that take advantage of spatial analogies. That means if the light switches are arranged exactly like the lights that they control, there will be no difficulty in using them. Besides the spatial mappings, there are also cultural,

biological, and perceptional ones. An example of cultural mapping would be reading. In most countries, people read from left to right, but some read from right to left or even top to bottom. Therefore, knowing the user group, including their culture, is important. Biological mapping is used when using additive dimensions, for example, amount, volume, and brightness. A rising level is naturally perceived as more, whereas a falling level is perceived as less. A perceptional mapping is when a user perceives an alignment between an action and its effect. When a steering wheel is turned to the right, the car turns to the right, so the user to understand intuitively cause and effect. [17]

2.2.5 Constraints

Just as affordances and signifiers offer actions, some constraints limit possible actions. To enhance natural mapping, Norman introduces constraints in four different areas. First, there are physical constraints. They are derived from the physical properties of an object. For example, a USB plug can only be plugged in one way because of its shape. Next, Norman explains cultural constraints. These are based on cultural standards, such as red representing stop or danger. Consequently, cultural constraints are not universal but are limited to certain cultures or regions. Then there are logical constraints, which use logical relations and principles. If something is assembled and a part remains, this suggests to the user that it was not assembled correctly. The last category of constraints is semantics. It refers to interpreting symbols, signs, and language to convey information and guide user actions. For example, a trash icon represents deletion. [17]

2.2.6 Feedback

While users interact, e.g., with an app, they need feedback such that the users understand that the app received the interaction and is processing it. Feedback communicates the current state, success, or failure of an action. It should be immediate and informative. If the time for feedback is too long, people may give up and move on to other things. Additionally, the provided feedback needs to be informative and prioritized so the users understand the provided feedback and recognize the most important feedback first. [17]

2.2.7 Knowledge in the World and in the Head

Norman uses the concepts of knowledge in the world and knowledge in the head to describe the interplay between human cognition and the surrounding world. These ideas highlight how external information and design may enhance human cognition while facilitating the cognitive strain on humans. Information or knowledge present in the environment or in artifacts but external to the user refers to knowledge in the world. It serves as an external memory, allowing the user to offload cognitive tasks onto the environment. In contrast, users' images or information in their minds refer to knowledge in the head. Knowledge in the head represents the user's internal

knowledge, competencies, and capabilities. Creating a seamless interaction between the user's knowledge in the head and the knowledge in the world can reduce the cognitive load on users and enhance their performance. [17]

2.2.8 C.R.A.P.

C.R.A.P. is an acronym for the four basic design principles: Contrast, Repetition, Alignment, and Proximity. These four principles provide a guideline for creating an attractive and effective design. Contrast is about creating visual differences be- tween elements in a design to highlight them. This can be done by changing different properties like colour, size, shape, texture, and typography. Creating variation within the design helps guide the user's attention and creates a hierarchy. Consistency is a key element in design and is achieved by repeating design elements throughout the design. This involves colors, shapes, fonts, or other visual elements. Using repetition helps to establish a sense of cohesion and organization. It adds visual interest and strengthens the overall visual impact of the design. The positioning of elements also matters and is explained by the Alignment principle. It refers to the arrangement of elements in a design along a visual axis or edges. It helps create order and structure, ensuring elements are visually connected and related. Suitable alignment improves readability, clarity, and overall visual balance. Aligning elements to a grid or using clear visual guides enhances the professional look of a design. Proximity is about grouping related elements to show their connection and thus creating a sense of organization. Placing elements that belong together closer to each other supports creating visual relationships and avoids clutter. Proximity helps users understand the hierarchy and flow of information within the design. [20]

2.3 User-centred Design

This project follows a user-centred design (UCD) approach to ensure optimal usability. Through the UCD approach, the developers gain a comprehensive understanding of the target users, including their capabilities, needs, expectations, goals, and the tasks required to achieve those goals and the physical and social environments in which they operate. [5]
Donald Norman, author of the book "The Design of Everyday Things" [17], describes this approach as an iterative process. First, Norman explains the observation phase, in which users are observed in their natural environment, such as at home, school, work, or social events, to gain a deep understanding of the user's persona. The second phase is called idea generation and focuses on creating many ideas without getting too attached to specific ones too soon. Creativity and exploration are key in this phase, and avoiding criticizing ideas or imposing constraints is important. Questioning everything, including fundamental assumptions, can lead to profound discoveries and solutions to problems. Challenging the obvious can often uncover new perspectives and innovative ideas. Next, the prototyping phase starts to validate an idea's feasibility. Quick prototypes can be made using simple tools like sketches, cardboard models, or digital

images. Prototyping allows testing and reveals important requirements. Additionally, prototyping serves to understand the problem better during the specification phase and to create real prototypes during the solution phase of the design process. [17]
These prototypes can vary in fidelity, with low- and high-fidelity prototypes serving different purposes at various stages of the design process. Low-fidelity prototypes are typically simple, quick to create and have low costs. They often consist of basic sketches, paper mock-ups or digital wireframes that visually represent the product's structure and functionality. In general, they do not offer any interaction. Low-fidelity prototypes are useful for exploring different design ideas in the early design stages. High-fidelity prototypes provide a more realistic representation of the final application. They are similar to the actual application in appearance, functionality and user experience. They include more interactions and functionalities. [8, 19]
Users test the prototype after an appropriate prototype is created in the third phase. Norman calls this the fourth phase testing. While testing, the developers gather different information starting from initial user feedback about the basic functionality of the design. Furthermore, they identify design flaws such as poor layout or unattractive visual design. As fidelity increases, the complexity of the gathered information increases as well. Users testing high-fidelity prototypes can identify usability issues and areas for improvement in the application and evaluate how well users can complete specific tasks within the application or the overall user experience. After completing the fourth phase, the process starts again from the beginning. By iterating through these phases, developers continuously improve requirements and prototypes and, thus, the final application. [17]

2.3.1 Iterations

Three different iterations are presented below to explain the distinct phases of the user-centred design approach. There may be several smaller iterations in between the iterations presented. Furthermore, the different prototypes are discussed.

Iteration 1 Observing the user was due to physical distance not possible. Instead, the users were interviewed as accurately as possible to collect all necessary information. Additionally, the users described their usual working steps to ensure nothing is missed. While gathering more and more information, the idea generation becomes more detailed.
With the help of the information gained, the first ideas were created, which were then presented through sketches (low-fidelity prototypes) with the users. Figure 1 shows one of the first paper prototypes created on a tablet. The home screen displays a list of created interviews and a button to create new ones. The list contains interviews with different users that have used this device. The interviews are password protected to prevent users from accessing interviews they did not create. The list elements contain brief information about the name of the interview and interviewer. Additionally, the user can filter the list by name, creation date or other criteria. The app header displays

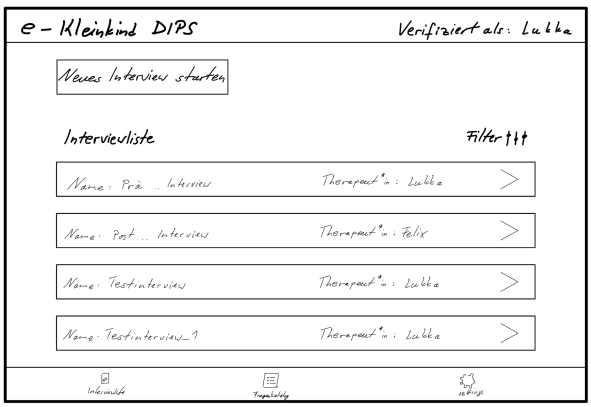
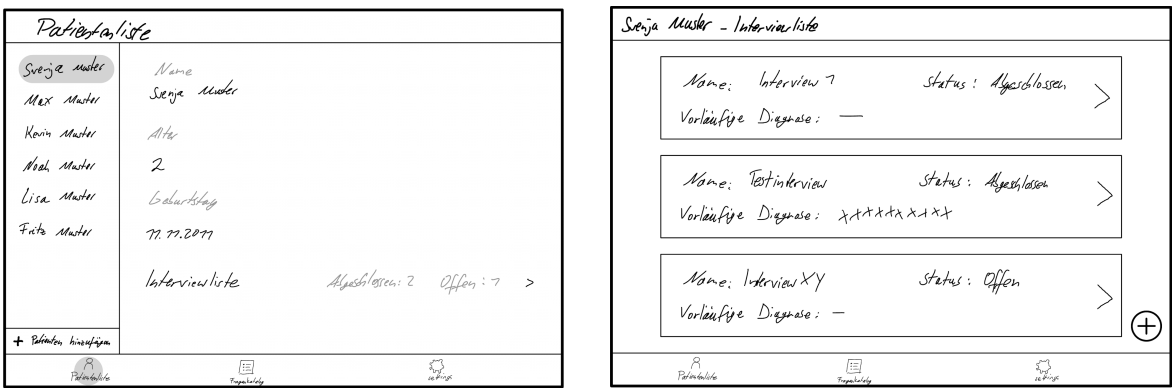


Figure 1: One of the paper prototypes of the first iteration of the home screen containing a list of created interviews by different users. Each interview is password protected to prevent unauthorized access.

the current page name and the username. In the bottom app bar, three different buttons are available for navigation between three pages. The first button is the current page. The second one takes the user to the overview of the question catalogue. There is no connection to a specific patient but only a general interview overview. The last one provides settings, e.g. to change the name or password.
Low-fidelity prototypes, particularly sketches, are static and do not immediately allow interactions. However, a setup was created such that some interaction was possible in the testing phase. During the meeting, the users could see the developer's screen and were instructed to speak out loud while interacting with the sketch. This approach follows the think-aloud method. Having users continually think aloud while using the system or sketches allows usability and other issues to be easily identified [9]. The Wizard of Oz prototyping approach is used to simulate interactions with paper prototypes. This means the user interacts with the paper prototype while the wizard (i.e. the designer) observes and operates the prototype behind the scenes. The wizard manually simulates the system's responses by changing or replacing prototype elements based on the user's actions or requests. This allows the designer to gather insights into how users interact with the design and identify usability issues. [4] For example, if a user wants to open the first interview in the list, the developer would simulate the user's click and open the corresponding sketch.

Iteration 2 Interviews and open discussions again replaced the observation phase to gain new insights into the purpose of the system and the problems faced by the users. This is followed by an approach that replaces the previous idea of one interview list for all users with user accounts containing only their data.
As seen in Figure 2a, the interview list was replaced by a patient list on the left. If an entry is picked, general information is displayed on the main page. The app header and the bottom bar keep their functionalities from the previous iteration. Clicking the list tile indicated by a right arrow at the bottom of the overview takes the user

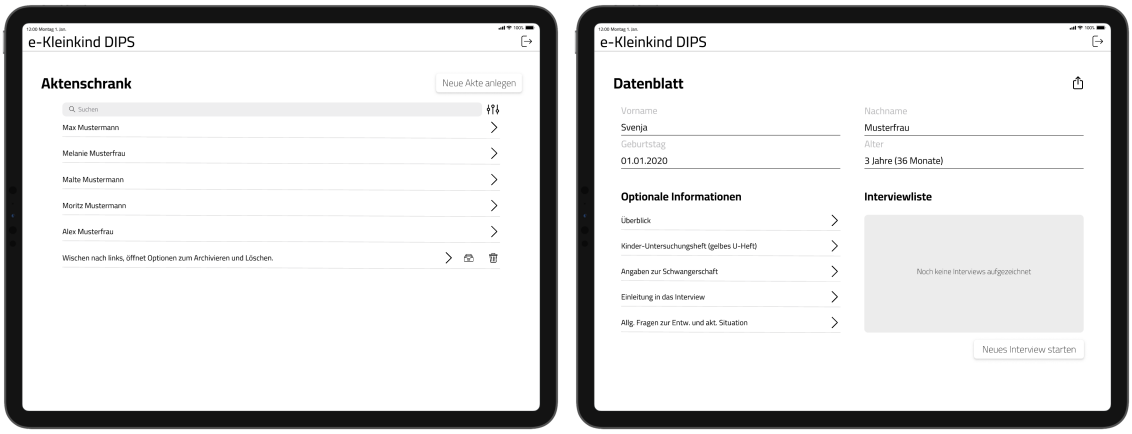


(a) (b)

Figure 2: (a) A list of patients only available to the logged-in user. The right area shows information about the currently selected entry in the list.
(b) Clicking on the tile *Interviewliste* will open a new page showing a list of the patient's interviews.

to the interview list on a new page, as shown in Figure 2b. Each entry represents an interview and displays the interview's status and, if available, a diagnosis.
As in the first iteration, the think-aloud method tests the prototype and collects appropriate user feedback.

Iteration 3 Through interviews and open discussions, all areas of concern of the paper-based interview guide and the current paper prototype were again considered leading to new ideas. The prototype's fidelity increases by gathering more information with each iteration. Therefore, a more interactive and realistic prototype is created using Figma. As illustrated in Figure 3, a click prototype was created. Figma provides a realistic device frame to encourage the user to interact. On the home page is the document cabinet, i.e., the list of all created patients of the current user. A swipe gesture to the left enables further actions. One is to delete the patient's document, and the other is to archive the document. Archiving refers to tidying up the list by having two separate lists of current and past patients (see Figure 3a). If the user wants to create a new patient, he clicks the button in the upper right corner redirecting him to the new page, as shown in Figure 3b. The user enters the patient's most important and obligatory data in four text fields in the upper section. In the left lower corner is a list of optional information. Entering these pages, text fields request the optional information of the corresponding category. The interview list is visible next to the list of optional information, which provides a grey container if it is empty. Otherwise, it would be filled with list tiles similar to the optional information but presenting information about interviews. If the user wants to start a new interview, he clicks on the button below the list and gets to the interview questions.
Users can test more freely and provide feedback without much effort by creating click prototypes and especially within Figma. Users are invited to the corresponding Figma



(a) (b)

Figure 3: (a) Click prototype created in Figma presenting a list of patient documents available to the logged-in user.
(b) This page appears after the user clicks the button to add a new patient document on the right above the list.

project and have access at any time to start the simulation of the click prototype. When using the click prototype, the user can provide feedback directly to the respective state via drag-and-drop speech bubbles. Furthermore, in Figma, among other things, UI elements can be linked with special actions and effects. This means that, e.g. a button can be linked to a click action, and after that, the page changes. In this case, navigation from the document cabinet (see Figure 7a) to the page for creating a new patient (see Figure 3b) is simulated. In the beginning, concrete click paths are created, also called scenarios, because only certain UI elements are linked to actions that prevent the user from interacting with anything else, although corresponding UI elements are visible. Increased fidelity increases the number of possible actions, resulting in more detailed scenarios.

3 Software Construction

This application aims to simplify diagnostic interviews and improve them economically and environmentally. First, it is explained how structured interviews based on the DIPS approach are used at the Ruhr University Bochum. Then the resulting basic requirements forming the foundation of the system are described. From this, the functionalities and design decisions regarding the software and the data model can be explained in more detail.

3.1 Structured Interviews

The DIPS are structured interviews to diagnose mental disorders. There exist several DIPS for specific target groups. In this project, the Kleinkind-DIPS is developed,

which is used to identify disorders in infants up to age six. Diagnostic criteria are tested based on the DSM-5 (Diagnostic and Statistical Manual of Mental Disorders)[14] to determine whether a mental disorder is present (lifetime diagnoses, comorbid diagnoses) or absent. The DSM-5 is the fifth version of a manual for assessing and diagnosing mental disorders.
The interview guideline is divided into several sections, each representing a mental disorder. In order to test the diagnostic criteria, a section contains several questions. These questions can be classified as either criterion or non-criterion questions. In the case of criterion questions, a letter refers to the corresponding diagnostic criterion based on the DSM-5. Non-criterion questions are advanced questions that may be relevant to therapy or allow for a more detailed analysis of the problem. There are 29 sections in total. Of these, four sections are independent of a disorder but serve as an introduction or screening into the interview.
Some sections depend on the age of the patient. For example, some sections require the patient to be younger than two years old. Other sections do not begin until the patient is four years old.
Most sections have jump rules that, when satisfied, exit the section and continue with another. These jump rules are usually located at the beginning of the section and are closely linked to the criterion questions. After the first asked criterion questions of the section, the interviewer can already determine whether the disorder can no longer be satisfied. In this case, a jump rule is fulfilled, the section is left and continued with the next one. But in some cases, the only statement that can be made is whether the disorder is not fulfilled in the present, but it may have been fulfilled once in the past. In this case, the section is not left, instead the interviewer has to rephrase and ask questions into the past tense.
In addition to the typical yes-no questions, there are other types. Some specific questions need a description in order to be answered. Other questions only allow a predefined (multiple) selection of possible answers. An exceptional type of question is the scale question. A scale question consists of different areas that contain a set of questions. Each area corresponds to a symptom. A symptom is fulfilled if a certain number of questions from the corresponding area are fulfilled. Overall, the scale question is fulfilled when a certain number of symptoms are fulfilled.
At the end of most sections, suffering and impairment are determined using the same eight questions. Suffering and impairment are considered to be fulfilled if certain questions of these eight questions are answered with a certain value. Most disorders are only considered to be fulfilled if suffering and impairment are also fulfilled. That means suffering and impairment can be both diagnostic criterion and non-criterion questions. If the interviewer is able to determine suffering and impairment and the disorder is met, he proceeds to the next section. If the disorder is not fulfilled, there is the option of asking the section again in the past tense. For this purpose, a question and corresponding jump rule take the interviewer back to the beginning of the section. From there, the initial criterion questions must be skipped and finally continue with

[14]https://www.psychiatry.org/psychiatrists/practice/dsm

the actual questions, which must be rephrased into past tense.
At the end of the interview, the diagnosis is determined. For this purpose, it is checked section by section whether all conditions are fulfilled so that the disorder may be assigned. For this, criterion question by criterion question is checked and it is determined whether it is fulfilled or not. In addition, questions within specific questions, such as scale questions and suffering and impairment, must be checked individually to assess whether they are met as a whole. Finally, the diagnosis can then be made. This is divided into a primary, any number of additional and previous diagnoses. A diagnosis has a severity level as well as its name. The age at the start and end time must also be entered for the previous diagnoses.
With the completion of the diagnosis, the interview is finished. Sometimes it is of interest, e.g., for research purposes, to transfer the data into a statistics program. For this purpose, the answers to the corresponding questions are entered manually into an Excel template. The statistics program may then read this Excel file.

3.2 Requirements

First, the system serves the purpose of guiding the user through a structured interview. The application focuses on a simple and clear construction of the interview content. During the interview, a stopwatch automatically records the time. In addition, a chapter overview helps for fast navigation and an integrated notepad enables the user to take notes without losing the focus on the interview content. After the interview, the system provides a list of all fulfilled disorders so that the user can determine a diagnosis without checking section by section in the interview content again. In addition to the interview, other general patient-related data can be recorded. This information, including all interviews, can be exported to a specific server the admin sets up. Usually, a user has more than one patient at a time, which is why the system features the management of different patient documents. Furthermore, several users likely share one device in a practice. Therefore, the system also offers user management. This leads to role management, in which the admin has certain additional rights, including assigning patient documents to other users in case a user leaves the practice, in addition to setting up the server.

3.3 Project Structure

The initial project is built with Flutter's command *flutter create app_name*. This creates a bunch of predefined files and directories. Most important is the file *pubspec.yaml* and the folder *lib*. The primary objective of the *pubspec.yaml* file is to manage the project dependencies and configurations. It acts like a manifest file, enabling developers to define the required packages. Every declared package within this file is automatically fetched and integrated during the build process. Furthermore, the file allows developers to specify various project-specific details and configurations, such as the project name, version, and description. Lastly, the developers can add paths to additional assets like

images within the file, ensuring their integration and accessibility inside the project.[15] Inside the folder *lib* exist all Dart code required by the application. The file *main.dart* is located on the first level. It serves as the entry point and the initial configuration file for the application. It sets up the necessary environment and defines the root widget. Furthermore, there are files representing the basic pages of the application, including the homepage, imprint, settings, profile, user management, and document cabinet. The homepage is the first visible screen after the launch image. It offers the user to log in, register or view the imprint. After the user is logged in, the document cabinet with the list of patient documents is presented. The user can switch between this page and the profile, settings, and user management in the bottom app bar. The user can change his data on the profile page, like e-mail or password. The settings and user management are only available for the administrator and offer the possibility of setting up a server for data export and (re-) assigning patient documents to users.
Furthermore, the *lib* folder includes subfolders (*data*, *widgets*, *dialogs*, *helper*, *interview*, and *document*) representing different functional areas of the application. This hierarchical structure allows for further organization and categorization of code files, enhancing clarity and ease of navigation within the project. All files related to the application's data are contained in *data*. Both the subfolders *widgets* and *dialogs* contain all reusable widgets, whereas reusable dialogs are separated. There are a variety of functions in *helper* that are not directly linked to widgets but rather conduct computations, validations, and formatting. The subfolders *interview* and *document* include the files for the two main areas: interviews and patient documents.

3.4 Data Model

As already introduced in 3.3, the folder *lib* includes the subfolder *data* containing all relevant files regarding the application's data. It is divided into static and non-static data. Static data is all that users cannot change (e.g., sections and their questions), while non-static data, such as patient information or interview answers, can be changed. A data model always has the same structure. It consists of two classes. The first class operates directly with the SQLite database and offers functions such as fetching, creating, updating, and deleting database entries. Moreover, the second class converts the fetched database entries into corresponding objects of the Dart class and vice versa. This helps structure and organize the data, leverage the benefits of object-oriented programming, simplify data manipulation and management, and ensure type safety. The following explains the database manager and the essential data models.

3.4.1 SQLHelper

The *SQLHelper* is not a data model but manages the basic functionalities of the database and controls all data models together. The most important function is *connectDatabase()*. It is used to connect to the SQLite database. It returns a *Future<Database>* object indicating that the connection is asynchronous and resolves

[15] https://docs.flutter.dev/tools/pubspec

to a database object once the connection is successfully created. Another function is called inside the function, which takes the following argument. First, the path to the database file is passed by joining the path to the application's database directory with the database file name. Another parameter specifies the version number. The database is password protected to prevent unauthorized access. Additionally, the function takes three different callbacks. One callback is only triggered when the database is created the first time and executes *createTables(...)* and *insertStaticData(...)*. By this, the tables are created and the static data is inserted. The next callback is always triggered and activates foreign keys. Finally, a callback that updates the static data if there are changes is executed. The function *connectDatabase()* ensures that the database connection is established, necessary tables are created, static data is inserted, and any additional setup is performed before returning the database object.

3.4.2 Data Model: Question

As mentioned above, the data model is divided into two separate classes, *QuestionModel* and *Question*. The *QuestionModel* represents a model for interacting with the SQLite database table *QUESTION*. Since the questions are the same for everyone and cannot be modified by the user, it is a static table. Therefore, the table name is written in capital letters to make distinguishing static from non-static tables easier.

The main purpose of the *QuestionModel* is to create the database table and insert data. The *createStaticTable(...)* function is called from the *SQLHelper* and takes the database object that is created inside the *connectDatabase()* function (see 3.4.1). Listing 1 shows only lines of the SQL statement in the corresponding function, which are also explained in more detail. The interview guideline consists of several sections, and each containing questions. Each question is unique and can be uniquely identified by the primary key *questionID* (see Listing 1, line 3). In order to assign a question to a section, a foreign key relationship is defined to the primary key of the database table *SECTION* (see lines 4 and 19). Apart from the text and numbering of the question, they also differ on other levels. Some questions require only a yes or no answer, and others offer predefined options. Therefore, the *QUESTION* table holds a foreign key of the *QUESTION_TYPE* table, assigning each question a specific type. Depending on the type of question, appropriate input options are displayed. For example, in the first case, there are only two checkboxes for yes and no. In the second case, there is a dropdown to select the answer. The different input options for the answers have the consequence that additional information is desirable for corresponding question types. For example, labels can be explicitly defined for the text input fields (see line 11) or a list of dropdown options (see line 10). Also, there are questions that are only relevant up to a certain age. For this, an age limit defines when the question should be hidden (see line 16). In this case, "hidden" means that the question is greyed out and only a teaser (the first line of the question) is displayed. Since the sections are also queried in the past tense, there is still a corresponding counterpart in the past tense for each question in the present tense (see line 17).

The most important property is whether the question is a criterion question (see line 9). Depending on this, the answer matters in the respective section or not. The rough functionality of jump rules is already explained in 3.1. To keep this functionality and improve this by applying jump rules automatically, special properties are added to the question model to identify these initial criterion questions, which determine whether a section is fulfilled in the present, in the past, or not at all. They are called *criterionPresent* (A) and *criterionPast* (V) (see lines 13 and 14). At the beginning of (almost) every section, there is always at least one criterion question A and one criterion question V. These two properties, which are set at the corresponding questions, are crucial for the execution of the jump rules (see 3.4.3). For simplicity, it is assumed that there is exactly one criterion question each of the present and the past. Then, there are three possible combinations: First, criterion questions A and V have been answered with no. This means the disorder cannot be fulfilled and has never been fulfilled. Second, criterion question A received a no response, whereas the answer to criterion question V was positive. Then the disorder is not fulfilled in the present. Thus it is determined whether the disorder has ever existed by looking at the past. Moreover, the last possibility is that criterion question A was answered with yes. Then the answer to criterion question V is irrelevant, because the disorder is potentially fulfilled in the present, and therefore, the past is no longer important. However, in the process of the section, it might turn out that the disorder is not fulfilled in the present, such as the past should be checked. For this, a flag exists to show the special question, allowing the user to decide if he wants to check the past or continue with the next section.
Other special flags can and must be set to ensure correct functionality. Exactly one question of a section must be marked as the first question of the present so that the jump rules automatically jump to the right place if the criterion questions about the past are skipped.
Each section always has a small set of questions at the end, which, considered together, decides whether suffering and impairment are fulfilled. If the given answers show that suffering and impairment are fulfilled, then the user has to explicitly decide about the assignment. The questions about suffering and impairment and the question about assignment have special flags. In the same way, if the criteria are met and thus the disorder can be assigned, the user is explicitly asked to decide with a corresponding question.
Lastly, there are scale questions. These are related questions that do not need to be fulfilled individually. Instead, it is checked whether a certain constellation exists for the set of questions (i.e., the scale question) to be fulfilled. A scale question is often divided into several categories (see line 12). For each category, it is defined how many questions must be fulfilled for the category to be fulfilled. Overall, the scale question is then fulfilled if a certain number of categories is fulfilled. For this, questions can be assigned to the categories respectively scale questions.

```
1   Future<void> createStaticTable(Database database) async {
2     await database.execute('''CREATE TABLE QUESTION(
```

```
3         questionID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
4         fkSectionID INTEGER,
5         fkQuestionTypeID INTEGER NOT NULL,
6         ...
7         questionText TEXT NOT NULL,
8         numbering TEXT NOT NULL,
9         criteriaLetter TEXT,
10        dropdownValues TEXT,
11        label TEXT,
12        category TEXT,
13        criterionPast INTEGER NOT NULL DEFAULT 0,
14        criterionPresent INTEGER NOT NULL DEFAULT 0,
15        ...
16        ageLimitInMonths INTEGER,
17        past INTEGER NOT NULL DEFAULT 0,
18        ...
19        FOREIGN KEY(fkSectionID) REFERENCES SECTION(sectionID),
20        ...
21      );''');
22    }
```

Listing 1: This function creates the table *QUESTION* in the database.

Besides the function to create the database table, the class *QuestionModel* also has the task, since it is a static table, to fill the table with data accordingly. For this purpose, each section has a file containing the corresponding questions. This brings a structure and overview to a large number of questions. The questions of all files are then inserted into the table in the function *QuestionModel.insertStaticData(...)*. This call occurs within the *insertStaticData(...)* function inside the *SQLHelper*.
In addition to creating the table and inserting static data, a data model can contain any number of other functions. In this case, Listing 2 shows a function that returns all stored questions in the database. First, line 2 connects to the database using the already known function *connectDatabase()* (see 3.4.1). After establishing the connection, an implicit SQL statement is executed on the database object (see lines 3 and 4). This results in a list of maps which is iterated over and a corresponding *Question* object is created.

```
1   static Future<List<Question>> getQuestions() async {
2     final db = await SQLHelper.connectDatabase();
3     final List<Map<String, dynamic>> maps =
4                 await db.query('QUESTION');
5     return List.generate(maps.length, (i) {
6       return Question.fromMap(maps[i]);
7     });
```

```
8     }
```

Listing 2: This function fetches all questions in the database table *QUESTION*. It transforms table entries into Dart objects of the class *Question* and returns a list of these.

This object belongs to the *Question* class representing the second important class of the data model. As mentioned above, it handles the conversion of the fetched data into corresponding objects of the Dart class to simplify data manipulation and management and ensure type safety. The data retrieved from the database comes in a list of key-value pairs, where the keys are strings and the values can be of any type (see Listing 2, line 3). Each map in the list represents a row of data from the database table, where the keys are the column names and the values are the corresponding values from the database. It is iterated over this list of maps using *fromMap(...)*. Listing 3 shows the simplified function which takes a map, i.e. the current element of the list of maps, and returns a *Question* object. Each question has a specific type and is represented by the foreign key relationship. Instead of storing only the foreign key in the object, a matching mapping is performed directly between the foreign key and the enumeration *QuestionType*. For this, a *QuestionType* variable *questionType* is declared and a default value is assigned. Then a switch statement determines the correct value (see Listing 3, lines 4-13). Another mapping between SQLite and Dart can be seen in lines 15 et sqq. in Listing 3. Line 10 in Listing 1 reveals that *dropdownValues* is of type *TEXT*. This property is used for the predefined answer options for questions with dropdowns. Since SQLite does not support a list of texts or numbers, a mapping must be done here. For this, the text from the database is always separated by a semicolon and assigned to a list of strings. Another reason for mapping are boolean values. SQLite does not support boolean values. Thus only flags are used, i.e., the value is 0 or 1. Line 25 shows how the current value of the map is determined with the ternary operator. If the value corresponds to a 1, true is set, otherwise false. The function then creates a new *Question* object using the values extracted or calculated from the passed map (see Listing 3, lines 20 et sqq.). They are provided as arguments to the constructor of the *Question* class to initialize a new object.

```
1   static Question fromMap(Map map) {
2     QuestionType questionType = QuestionType.yesNo;
3
4     switch (map['fkQuestionTypeID']) {
5       case 1:
6         questionType = QuestionType.yesNo;
7         break;
8
9       case 2:
10        questionType = QuestionType.description;
11        break;
12        ...
13    }
```

```
14
15      List<String> dropdownValues = [];
16      if (map['dropdownValues'] != null) {
17        dropdownValues = map['dropdownValues'].split('; ');
18      }
19
20      return Question(
21        map['questionID'],
22        questionType,
23        map['questionText'],
24        map['numbering'],
25        map['aboutSufferingImpairment'] == 1 ? true : false,
26        ...
27        dropdownValues,
28        ...
29      );
30    }
```

Listing 3: This function converts one entry of the database table *QUESTION* into an object of type *Question*. Different types are mapped between SQLite and Dart accordingly.

3.4.3 Data Model: Jump Rule

This data model is also divided into two classes, *JumpRuleModel* and *JumpRule*. The capitalized table name in Listing 4 indicates that the jump rules are static.

Listing 4 shows the function to create the *JUMP_RULE* database table. The interview content is not static but adapts to the answers given. There are jump rules to decide which questions or even whole sections should be skipped. A jump rule is executed only within a section. Therefore, a foreign key relationship to the static table *SECTION* is defined (see Listing 4, lines 4 and 14). Furthermore, a start question and a next question, or a next section are required (see lines 5-7). The questions between the start and end question are then hidden or displayed again accordingly. If a section is specified instead of the end question, all questions beginning from the start question are hidden or displayed again. Originally, not the numbering (see Listing 1, line 8) of the questions, but the primary key *questionID* was defined here. However, it was changed to numbering to simplify the maintenance of the jump rules. If the primary keys were used and any question before them were removed, the primary keys for the start and end questions would have to be manually decreased by one. Since, especially during active development, the set of questions was not final, this led to unnecessary additional work. Because of this, the numbering of the questions is now used here, which is also unique per section. In the corresponding Dart class *JumpRule*, the correct primary keys dynamically replace the numbering. There is *sqlStatement* to enable

a jump rule to determine when a question or section can be skipped (see Listing 4, line 8). Imagine there is a question 7, which can be answered with yes or no. In addition, question 7.1 exists as a follow-up question. In case question 7 was answered with no, there is no need to ask and answer question 7.1 anymore. A SQL query can map this dependency, as seen in Listing 5. The strings between "<" and ">" are replaced with the corresponding IDs whenever the jump rules are fetched from the database. In the given case, the query result should be one since the answer exists and was answered with no, i.e., *answer.yesNo* is 0. The jump rules need a corresponding expected result to validate whether the result is correct (see Listing 4, lines 9-11). If the query result meets this defined expected result, question 7.1 is skipped and continues with the next question. By using SQL queries, it is possible to map arbitrary conditions. Thus, they are also used to validate scale questions by setting the flag *isScale* (see line 13). In the case of only one category within a scale question, the SQL query is similar to Listing 5. Scale questions having more than one category and each category has its condition, nested SELECT-statements can be used to validate whether the scale question is satisfied or not. Since the questions have a corresponding counterpart in the past tense, the jump rules can also be defined for the past by setting a flag (see line 12).

```
1   Future<void> createStaticTable(Database database) async {
2     await database.execute('''CREATE TABLE JUMP_RULE(
3       jumpRuleID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
4       fkSectionID INTEGER NOT NULL,
5       startQuestionNumbering TEXT NOT NULL,
6       nextQuestionNumbering TEXT,
7       nextSectionTitle TEXT,
8       sqlStatement TEXT NOT NULL,
9       countShouldBe INTEGER,
10      countShouldBeLTE INTEGER,
11      countShouldBeGTE INTEGER,
12      past INTEGER NOT NULL DEFAULT 0,
13      isScale INTEGER NOT NULL DEFAULT 0,
14      FOREIGN KEY(fkSectionID) REFERENCES SECTION(sectionID)
15    );''');
16  }
```

Listing 4: This function creates the static database table *JUMP_RULE*.

```
1   SELECT COUNT(*) AS result FROM answer
2     WHERE fkInterviewID = <fkInterviewID>
3     AND fkQuestionID = <fkQuestionID>
4     AND yesNo = 0;
```

Listing 5: This function illustrates how the arbitrary conditions of jump rules are mapped in one property. In this case, the jump rule checks whether a specific question is answered with no.

In 3.4.2, the three combinations of criterion questions A and V are introduced, as well as the fourth option in which criterion question A was answered with yes. However, the disorder is still unsatisfied; therefore, the section should be queried in the past tense. In the beginning, specific jump rules were in each section to check these different answer combinations. Nevertheless, with a few exceptions, creating a pattern for these four options was possible. Through the pattern and an appropriate handling of the exceptions it was possible that the number of jump rules could be greatly reduced, which also decreases the effort for maintenance. This pattern uses a combination of different flags on various questions within a section. It is explained in 3.5.4. Since these jump rules do not exist in the individual sections but are created the same for each section, they are called static jump rules.

Similar to the *QuestionModel* in 3.4.2, the *JumpRuleModel* also has additional functions, e.g., inserting and fetching jump rules. A special feature is the function *executeJumpRule(...)* (see Listing 4).
It takes a *JumpRule* object and returns an updated version; more specifically, it evaluates the result of the SQL statement to determine whether the jump rule is satisfied. First, a connection to the database is established, then the SQL statements of the static jump rules are executed, i.e., the dynamic jump rules are considered separately. Afterwards, it is checked if the result is empty (see Listing 6, lines 7 et sqq.). In this case, the result is set to false and the function is exited. Otherwise, the result is converted to an integer (see lines 12 and 13) and compared to the predefined result of the *JumpRule* object (see lines 15 et sqq.). Usually, it is sufficient if the query result of the jump rule is equal to the predefined result. However, for the jump rules of the scale questions, it is important to check whether the result is greater or equal to the predefined result. This is because it is not the number of fulfilled symptoms counted but the number of unfulfilled symptoms. From this, it can be determined whether it is still possible to fulfill the scale question at all. For example, a scale question is fulfilled according to the interview guideline as soon as at least three of the 15 possible symptoms are given. Conversely, this means that if 13 or more symptoms are not fulfilled, the scale question cannot be fulfilled anymore and the section is skipped due to a jump rule. This behaviour cannot be mapped exclusively with *countShouldBe*, so *countShouldBeGTE* and *countShouldBeLTE* have been added. Finally, the updated *JumpRule* object is returned.

```
1   Future<JumpRule> executeJumpRule(JumpRule jumpRule) async {
2     final db = await SQLHelper.connectDatabase();
3     List<Map<String, dynamic>> queryResult =
4           await db.rawQuery(jumpRule.sqlStatement);
5
6     // return false if queryResult is empty
7     if (queryResult.isEmpty) {
8       jumpRule.queryResult = false;
9       return jumpRule;
```

```
10      }
11
12      int count = int.parse(
13              queryResult[0]['result'].toString());
14
15      if (jumpRule.countShouldBeGTE != null) {
16        jumpRule.queryResult =
17              count >= jumpRule.countShouldBeGTE!;
18      } else if (jumpRule.countShouldBeLTE != null) {
19        jumpRule.queryResult =
20              count <= jumpRule.countShouldBeLTE!;
21      } else {
22        jumpRule.queryResult = count == jumpRule.countShouldBe;
23      }
24
25      return jumpRule;
26    }
```

Listing 6: Illustration of how jump rules are executed.

The corresponding Dart class *JumpRule* converts the retrieved database entries and the respective objects. In doing so, it proceeds similarly to Listing 3 in 3.4.2. The *fromMap(...)* function in this class differs because the jump rule templates (see Listing 5) defined in *sqlStatement* must be converted to real SQL statements. In the first place, the numbering for the start and next questions is replaced by their actual primary keys. If the database entry for *nextQuestionNumbering* is null, the primary key of the section is searched accordingly and entered in *JumpRule.nextSectionID*. Furthermore, when mapping to the Dart object, an additional property is added which originally does not exist in the database table. This is *queryResult*, which represents the result of the execution of the SQL statement of the respective jump rule. It has already appeared in lines 8 or 16 et sqq. in Listing 6 and stores the result of the executed SQL statement.

3.4.4 Data Model: Answer

The *AnswerModel* is a non-static database table. Therefore, its name is written in lowercase. Listing 7 presents the function of creating the database table.

Each answer can be assigned to exactly one question. Therefore, a foreign key relationship to the primary key of the database table *QUESTION* is added in lines 4, 14, and 15. Additionally, to assign an answer to the correct interview, a foreign key of the database table *interview* is added (see Listing 7, lines 5, 16, and 17). As introduced in 3.1, there are different types of questions and consequently, there must be different types of answers. However, instead of creating a separate model for each option, one model that covers all answer options. The simple cases are questions that allow only

one answer option. For example, these are yes-no questions. As mentioned earlier, SQLite does not support boolean values, instead 0 or 1 are entered here accordingly (see Listing 7, line 6). For some questions, the answer is a description. Therefore, there is the property *description*. Furthermore, there is the possibility to choose one or more answer options from a predefined set (see lines 8 and 9). Some questions only allow a number, a time, or a duration as an answer. At the end of each section, there is a question for the interviewer to decide whether to assign the disorder. This question only appears if the application has calculated that all the criteria for fulfilling the disorder are given. The interviewer then decides between assigning, not assigning or not sure. Finally, one special question that shows an empty diagram and the user can draw points that are automatically connected with a line resulting in a mood curve. Since there is a column for each possible answer type, any combination of answers to a question can also be stored. For example, there are questions to which the answer is yes or no and additionally, a description is necessary.
In addition to the function for creating the database table, *AnswerModel* also provides the usual functions for creating, fetching, updating and deleting database entries.

```
1   Future<void> createTable(Database database) async {
2     await database.execute('''CREATE TABLE answer(
3       answerID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
4       fkQuestionID INTEGER NOT NULL,
5       fkInterviewID INTEGER NOT NULL,
6       yesNo INTEGER,
7       description TEXT,
8       dropdownValue TEXT,
9       multiSelection TEXT,
10      number TEXT,
11      timeOrHours TEXT,
12      assignDisorder INTEGER,
13      moodCurvePoints TEXT,
14      FOREIGN KEY (fkQuestionID) REFERENCES
15            QUESTION(questionID),
16      FOREIGN KEY (fkInterviewID) REFERENCES
17            interview(interviewID) ON DELETE CASCADE
18    );''');
19  }
```

Listing 7: This function creates the non-static table *answer* in the database.

Since the *answer* table is not static, a new function called *toMap(...)* is needed in addition to the already known *fromMap(...)*. It performs exactly the opposite tasks, i.e. instead of converting the fetched data into objects of the *Answer* class, *toMap(...)* converts the *Answer* objects back into a map. *toMap(...)* returns a Map<String, dynamic> object that contains key-value pairs. Listing 8 shows the simplified function to convert an *Answer* object into a map representation. First, a nullable integer variable

yesNo is declared. It is assigned either 0 if the answer to the yes-no question is no (i.e. false) or 1 if the answer is yes. If *this.yesNo* is null because it is not a yes-no question at all, the variable remains null (see Listing 8, lines 2-5). In the same way, the variable *this.multiSelection*, which contains a list of strings, must be converted into a single string. To do this, the elements are joined with a semicolon (see lines 7-19). To conclude the section, if the criteria are met, the interviewer is asked if he would like to assign the disorder. Three options exist, represented by an enumeration within the Dart object. For the SQLite table, the enumeration is replaced by an integer value (see lines 21-34). Another property that needs to be converted is the mood curve. In the *Answer* object, it is represented by a list of x and y coordinates. This list of points is converted to a single string within an outsourced function (see lines 36-40). This string is then of the form $x_0, y_0; x_1, y_1; ..., x_n, y_n$. Then a map object is created and populated with key-value pairs. The keys correspond to the property names of the *Answer* object, and the values are obtained from the corresponding properties. The returned map object can then be stored in the SQLite database (see lines 42 et sqq.).

```
1   Map<String, dynamic> toMap() {
2     int? yesNo;
3     if (this.yesNo != null) {
4       yesNo = this.yesNo! ? 1 : 0;
5     }
6
7     String? multiSelection;
8     if (this.multiSelection != null) {
9       multiSelection = '';
10      for (int i = 0; i < this.multiSelection!.length; i++) {
11        if (i == this.multiSelection!.length - 1) {
12          multiSelection =
13            multiSelection! + this.multiSelection![i];
14        } else {
15          multiSelection =
16            '${multiSelection!}${this.multiSelection![i]}; ';
17        }
18      }
19    }
20
21    int? assignDisorder;
22    switch (this.assignDisorder) {
23      case AssignDisorder.yes:
24        assignDisorder = 1;
25        break;
26      case AssignDisorder.no:
27        assignDisorder = 2;
```

```
28            break;
29          case AssignDisorder.notSure:
30            assignDisorder = 3;
31            break;
32          default:
33            assignDisorder = null;
34        }
35
36        String? moodCurveString;
37        if (moodCurvePoints != null) {
38          moodCurveString =
39              convertMoodCurvePointsToString(moodCurvePoints!);
40        }
41
42        return {
43          'answerID': answerID,
44          'fkInterviewID': fkInterviewID,
45          'fkQuestionID': fkQuestionID,
46          'yesNo': yesNo,
47          'description': description,
48          'dropdownValue': dropdownValue,
49          'multiSelection': multiSelection,
50          'number': number,
51          'timeOrHours': timeOrHours,
52          'assignDisorder': assignDisorder,
53          'moodCurvePoints': moodCurveString,
54        };
55    }
```

Listing 8: Presents a function which converts objects of type *Answer* into a map. This makes it easier to write them into the database.

3.4.5 Data Model: Disorder and Diagnosis

The system distinguishes between disorders and diagnoses, so there is a data model for each. Disorders primarily indicate the result of the respective section during the interview. In contrast, diagnoses represent the actual clinical decision of the interviewer. Both disorders and diagnoses are non-static.

Listing 9 shows the properties of the database table for the *DisorderModel*. Appropriate foreign key relationships are used for a unique mapping between *Disorder*, section, and interview (see Listing 9, lines 4, 5, and 9-12). The property *assignDisorder* stores the result of the section, whether the disorder was assigned, unassigned or marked with not sure. The decision happens explicitly at the end of the section by the interviewer

or implicitly by the interviewer if a jump rule is fulfilled. Consequently, the section is left (see lines 6-7). Since a disturbance can also occur in the past, the section can be queried in the past tense. The property *disorderPast* makes clear whether it is a current or a past disorder.

```
1   Future<void> createTable(Database database) async {
2     await database.execute('''CREATE TABLE disorder(
3       disorderID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
4       fkInterviewID INTEGER NOT NULL,
5       fkSectionID INTEGER NOT NULL,
6       assignDisorder INTEGER NOT NULL
7               CHECK (assignDisorder BETWEEN 1 AND 3),
8       disorderPast INTEGER NOT NULL DEFAULT 0,
9       FOREIGN KEY (fkSectionID) REFERENCES
10              SECTION(sectionID),
11      FOREIGN KEY (fkInterviewID) REFERENCES
12              interview(interviewID) ON DELETE CASCADE
13    );''');
14  }
```

Listing 9: This function creates the table *disorder* in the database.

On the other hand, some diagnoses that are created exclusively and consciously by the interviewer at the end of the interview. Listing 10 shows the function of the *DiagnosisModel* that creates the corresponding table in the database. For a unique assignment, the foreign key relationship to *interviewID* is used (see Listing 10, lines 11-12). A diagnosis can be selected via a dropdown (see line 6). However, the list of possible diagnoses is extremely long, so an additional free text field is provided (see line 7). In addition to the name, diagnoses also differ in their type. There is exactly one primary diagnosis and any number of additional and previous diagnoses. The type of diagnosis is mapped by SQL using integer values and by Dart using an enumeration (see line 5). In addition, diagnoses have a severity level (see line 8). Since previous diagnoses are already completed, two further properties are necessary to store the start and end times (see lines 9-10). An exact date is unnecessary, but the age in months is sufficient.

```
1   static Future<void> createTable(Database database) async {
2     await database.execute('''CREATE TABLE diagnosis(
3       diagnosisID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
4       fkInterviewID INTEGER NOT NULL,
5       diagnosisType INTEGER NOT NULL,
6       name TEXT NOT NULL,
7       other TEXT,
8       severity INTEGER CHECK(severity >= 0 AND severity <= 8),
9       start INTEGER,
10      end INTEGER,
```

```
11          FOREIGN KEY (fkInterviewID) REFERENCES
12                  interview (interviewID) ON DELETE CASCADE
13      );''');
14    }
```

Listing 10: This function creates the table *diagnosis* in the database.

3.5 Major Software Components

The application's core task is conducting structured interviews and assisting in making the clinical diagnosis. In order to be able to use the interview guideline optimally with the help of the application, an interaction of different components is necessary. These components are explained below. It is assumed that a user has successfully registered or logged in, created a patient document and started an interview.

3.5.1 InterviewManager

An interview may be accessed either by the interview card, i.e. the interview already exists, or by starting a new interview, or the *InterviewManager*. It serves as an orchestration of all functionalities needed during an interview process. It is a stateful widget since its content will change (see 2.1.2). This widget requires a couple of arguments to work, as Listing 11 shows. First, the information about the current patient must be passed, represented by an instance of *PatientDocument*. This information is mainly used to present patient data in the top bar and is passed down to child widgets to assign future data to the correct patient. Furthermore, the interview itself is an argument. Here information is included, such as the creation date, the interview duration, the interviewer's name or the interview's status. In line 4 a nullable variable *section* is defined. This variable is not needed initially, but once the *InterviewManager* calls itself when a new section is entered. Similarly, the variable *stopWatchWasRunning* is defined below it.

```
1   class InterviewManager extends StatefulWidget {
2     final PatientDocument patientDocument;
3     final Interview interview;
4     final Section? section;
5     final bool? stopWatchWasRunning;
6     ...
7   }
```

Listing 11: The arguments of the stateful widget *InterviewManager*.

The first step of a stateful widget is to initialize its state by executing *initState()*. It is a function of the Flutter State class and is overridden in the corresponding state class of a stateful widget. It is called when the widget is first inserted into the widget tree and allows for initialization tasks. First, the case is considered when the *InterviewManager* was opened the first time. The first step is to load the chapters, sections and

their questions from the database. This action is performed only once, i.e. if the manager is opened the first time. This avoids unnecessary database calls and saves some resources. Then the current section is determined by selecting the first section from the previously loaded data. At the end, the stopwatch is initialized and started. In this case, *section* and *stopWatchWasRunning* do not influence the manager. However, whenever the section changes, for example, due to a jump rule, a new *InterviewManager* is placed on the navigation stack. Then both arguments are relevant to display the correct content. In case of a jump rule, the system calculates which section follows. Usually, this is the following section, but rarely a section even further back. Also, the status of the stopwatch is passed. If the user has stopped the time, it will be stopped on the next page. As soon as a new instance of the manager is created, the variable *interview* is also updated. Among other things, the duration is overwritten with the current time or the status is changed. Always putting a new *InterviewManager* on the navigation stack allows easy navigation between pages without any additional effort.
As mentioned, the *InterviewManager* loads all chapters, sections and questions exactly once. This data is then passed to the child widget *ChapterOverview*. The chapter overview is opened as a drawer from the left by clicking on the left icon in the bottom bar. It is primarily used to provide an overview and enables quick navigation within the interview. Furthermore, the *InterviewManager* controls the notepad via another drawer on the right side. The notepad enables the user to take notes during the interview.
The most important task is the navigation between the sections. The function *pushNextSection(...)* is responsible for this, which considers various scenarios such as the drawer, jump rules and the floating action button (FAB). Listing 12 presents the simplified function, which takes a nullable ID of a section and a boolean indicating the origin of the function call. It does return nothing. Instead, it pushes a new instance of the *InterviewManager* onto the navigation stack resulting in an updated UI (see Listing 12, lines 13 et sqq.). First, the ID of the next section is determined by checking some conditions within if else-if statements (see Listing 12, line 3). Either the call comes from the drawer, *chosenNextSectionID* is set to ID of the clicked section. Alternatively, the call originates from the child widget *InterviewPage* due to an applied jump rule containing the appropriate section. The last option is that the FAB is clicked, setting *chosenNextSectionID* to the next available section. Next, all sections are searched to find the correct section based on the previously determined *chosenNextSectionID*. If successful, the drawer may be removed from the navigation stack (i.e. closed; see lines 9-11). Afterwards, the stopwatch is stopped and the current duration of the stopwatch overwrites the interview's duration. Finally, a new instance of *InterviewManager* is placed on the navigation stack (see lines 13 et sqq.). The new instance of *InterviewManager* contains updated arguments, such as the section and the interview, but also the status of the stopwatch (see lines 20 et sqq.). The time is stopped or continued on the next page, depending on the status (see line 24).

```
1   void pushNextSection(int? nextSectionID, bool fromDrawer) {
2     ...
```

```
3       int chosenNextSectionID = ...;
4       ...
5
6       for (var section in InterviewManagerHelper.allSections) {
7         if (section.sectionID == chosenNextSectionID) {
8           // close the drawer
9           if (fromDrawer) {
10            Navigator.pop(context);
11          }
12
13          Navigator.push(
14            context,
15            MaterialPageRoute(
16              builder: (context) {
17                stopwatchSubscription?.cancel();
18                updateInterviewStopwatchTimer();
19
20                return InterviewManager(
21                  patientDocument: widget.patientDocument,
22                  interview: widget.interview,
23                  section: section,
24                  stopWatchWasRunning: timerOn,
25                );
26              },
27            ),
28          );
29          ...
30        }
31      }
32    }
```

Listing 12: An excerpt of a function that handles the navigation between sections within the interview.

3.5.2 InterviewPage

The actual content, i.e. the questions and answers, are not controlled by the *InterviewManager*, but by the *InterviewPage*. The *InterviewPage* needs the arguments to display the correct section's questions and answers, as shown in Listing 13. In the first place, this is the current *section*, which contains all the questions (see Listing 13, line 2). Also, *interview* is needed to display the answers to the correct interview (see line 3). Furthermore, the function *pushNextSection(...)* is required in line 4 because the jump rules are evaluated in this widget. Lastly, the variable *patientDocument* is used, e.g. to hide questions if the age is exceeded or not reached (see line 5).

```
1   class InterviewPage extends StatefulWidget {
2     final Section section;
3     final Interview interview;
4     final Function pushNextSection;
5     final PatientDocument patientDocument;
6     ...
7   }
```

Listing 13: The arguments of the stateful widget *InterviewPage*.

The *InterviewPage* is a stateful widget, so its state is initialized first. For this, the jump rules of the section are fetched. In addition, all questions in the section are run through to find special questions and assign them to appropriate variables. These are, for example, questions marked as *criterionPresent* or *criterionPast* (see 3.4.2). Moreover, the different categories of the scale questions are collected, which are later used to evaluate them. Identifying these questions and categories is important as it determines how the section behaves in certain situations. At the same time, the variable *visibleQuestions* is initialized with key-value pairs. The keys are the questions' identifiers and the values correspond to a boolean that controls whether the question is shown or hidden. Furthermore, it is checked whether a database entry already exists in *Disorder* (see 3.4.5). In addition, it is checked whether the patient is of the recommended age. Otherwise, a dialog appears that allows the user to skip the section. Finally, it is determined whether the section's past should be asked. In this case, the section with questions in the past tense is added below the section (with questions in the present).

```
1   void updateInterviewPageContent() async {
2     await toggleModePast();
3     await executeJumpRules();
4     await checkShowQuestionSufferingImpairment();
5     await checkShowQuestionAssignDisorder();
6     await checkShowQuestionAboutPast();
7     ...
8   }
```

Listing 14: A function that executes a set of other functions whenever an answer is entered.

The most important function is shown in Listing 14 and controls several functions. It is called inside *QuestionManager* (see 3.5.3) whenever a response is given or already present.
The first function is the same one described in the previous paragraph and handles adding or removing the section with its questions in the past tense. Precisely, there are two possibilities when the past is activated. On the one hand, all criterion questions A must be answered with no and at least one criterion question V must be answered with

yes. On the other hand, the disorder is not fulfilled in the present, but the user wants to ask for the past explicitly and answers the special question with the corresponding flag with yes.
After the current mode is defined, all jump rules are executed (see Listing 14, line 3) that were fetched within the *initState()*. With the help of the jump rules, it is ensured that the correct questions are always visible or hidden. Jump rules are divided into static and dynamic ones and can be assigned to the past (see 3.4.3). The function *executeJumpRules()* in Listing 15 always executes the jump rules of the present (seeListing 15, lines 25 et sqq.) since this part of the section is always visible. Only if the past mode is enabled the jump rules of the past are executed as well (see lines 32 et sqq.). The jump rules are executed and their results are stored in *executedJumpRules* in line 2. Afterwards, some conditions, each representing a static jump rule, are checked. If a condition is met, the corresponding jump rule is executed and appended to the *executedJumpRules* list with its result (see lines 5 et sqq.). Here, the special questions found inside *initState()* matters. Finally, the individual jump rules are applied to the questions in a specific order. For this, the boolean of the corresponding question in *visibleQuestions* is set to the result of the skip rule. The order is important because the jump rules are not mutually exclusive. For example, one jump rule hides all questions up to the first question in the past section, but another jump rule would display one of the hidden questions again. Some jump rules would have the effect of displaying a dialog because the disorder may no longer be satisfied due to the input made. Then the user can decide if he wants to stay in the current section or jump to the next one. The forwarded function of the parent widget *pushNextSection(...)* is then called as a result.

```
1   Future<void> executeJumpRules() async {
2     List<JumpRule> executedJumpRules = ...;
3
4     // execute static jump rules
5     if (firstQuestionInPresent != null &&
6               criterionQuestionsPresent.isNotEmpty) {
7       executeJumpRules.add(...);
8     }
9
10    if (firstQuestionInModePast != null &&
11              criterionQuestionsPast.isNotEmpty) {
12      executeJumpRules.add(...);
13    }
14
15    if (criterionQuestionsPresent.isNotEmpty &&
16              criterionQuestionsPast.isNotEmpty) {
17      executeJumpRules.add(...);
18    }
```

```
19
20      if (aboutPastQuestion != null) {
21        executeJumpRules.add(...);
22      }
23
24      // apply jump rules results
25      if (executedJumpRules.isNotEmpty) {
26        for (var jumpRule in executedJumpRules) {
27          ...
28        }
29      }
30
31      // apply jump rules of past
32      if (modePast) {
33          for (var jumpRule in executedJumpRules) {
34            ...
35          }
36      }
37    }
```

Listing 15: An excerpt of the function that applies the results of the jump rules to the UI.

Besides showing and hiding questions, there are also special questions that are initially not rendered at all but only in certain cases. These are the questions about suffering and impairment, assigning the disorder, and whether the section should be asked in the past tense again. Each question is checked separately inside *updateInterviewPageContent()* (see Listing 14, lines 4-6).
The function *checkShowQuestionSufferingImpairment()* (see Listing 14, line 4) does not take any arguments or return any result. It performs an asynchronous task due to database calls and uses a SQL statement to validate whether the question about suffering and impairment should be displayed. First, the function checks if this question is in this section, both for the present and the past. This assignment is done in the *initState()* of the *InterviewPage* widget. The question exists only if suffering and impairment are a criterion. If so, the *DiagnosticToolHelper* is used to check if the correct answers are given. The same eight questions always query suffering and impairment. These questions are answered on a scale from zero to three, except for two. Suffering and impairment are fulfilled if at least one question is answered with at least two. As soon as the past is active, the same test is performed for the questions in the past tense as well.
As the answers are entered into the app, the app constantly checks whether the disorder is fulfilled. If this is the case, a question is displayed at the end, which informs the user that the disorder is fulfilled and also asks to decide whether the disorder should be assigned, not assigned or marked as not sure. The function *checkShowQuestionAssignDisorder()* is used for this purpose (see Listing 14, line 5). It takes no arguments

and returns nothing. Instead, it shows or hides a question in the UI. The function always checks on the one hand if the disorder is fulfilled in the present and on the other hand if the past mode is active and also if the criteria for a previous disorder are fulfilled. The *DiagnosticToolHelper* does the actual calculations. There the following conditions are checked in the form of guards. If a guard is fulfilled, the function is left directly. This saves unnecessary operations. The first condition is whether all visible criterion questions are fulfilled, i.e. answered with yes. "visible" refers to the variable *visibleQuestions*, which contains key-value pairs of all questions of the current section (See 3.5.2). This ensures that skipped criterion questions are irrelevant when evaluating the section. Next, the criterion questions for the present are tested. Of these, at least one must be answered with yes. This condition is only tested if the past is not active. Third, it is tested whether the interviewer explicitly assigned suffering and impairment. Last, the optional scale questions are validated using the appropriate skip rule. If the skip rule is met, the section is left and consequently, the section is not met. If all conditions are met, the section and, hence, the disorder is fulfilled. Finally, the question for assigning the disorder can be displayed.
The last special question is the question about the past, which is evaluated by the function *checkShowQuestionAboutPast()* (see Listing 14, line 6). It takes no arguments and instead of returning a result, the corresponding question about the past is displayed. It is asynchronous due to database calls. The function is split into two parts. The first part is only executed if suffering and impairment are a criterion since the question about assigning suffering and impairment is checked. If the user did not assign suffering and impairment, i.e. answered the question with no, the function will display the question about the past. The second part is only executed if the first part is not executed. The second part is only executed if the first part was not executed. Here the question about assigning the disorder is considered. The user must answer this question with no or not sure. Only then the question about the past appears.

3.5.3 QuestionManager

Each question on the *InterviewPage* is controlled by its own *QuestionManager* (see Listing 16). Its main task is to display the questions and their answers. It takes several arguments to perform all its tasks. For the *QuestionManager* to display the correct question and answer, a corresponding object of *Question* (see 3.4.2) is needed. Also, the state of visibility of the question is passed from *InterviewPage*, i.e. only a teaser or the full question is displayed. Since *updateInterviewPageContent()* is always executed as soon as an answer is entered or already present, the function is an argument of this widget (see Listing 16, line 4). The already introduced function *pushNextSection()* and the section identifier are necessary for one specific purpose, such that they are optional arguments. Last, the *interviewID* is needed to fetch the correct data.

```
1   class QuestionManager extends StatefulWidget {
2     final Question question;
3     final bool isVisible;
4     final Function updateInterviewPageContent;
```

```
5       final Function? pushNextSection;
6       final int? sectionID;
7       final int interviewID;
8       ...
9     }
```

Listing 16: The arguments of the stateful widget *QuestionManager*.

The *QuestionManager* is a stateful widget that initializes its state at the beginning. It checks if there is an *answer* in the database table *Answer* (see 3.4.4) for this question and interview. If there is an answer, the *answer* object is populated accordingly and passed to an appropriate child widget. Otherwise, *answer* will remain initialized with the default values.
The widget consists of a static and a dynamic component. The question text is static and is the same for all possible types of questions. Below the question are the input options, which depend on the type of question. If there is an answer to this question, it will be displayed according to the input fields. The correct child widget must be rendered for the correct input fields to be displayed. Each question type has a separate widget (yes-no questions, descriptions, dropdowns, multi selection, and more). Listing 17 shows the snippet of the switch statement within the widget's *build(...)* function. The switch statement takes the *InterviewPage* object passed by the *question* and evaluates the question type (see Listing 17, line 1). Then the appropriate case is executed. Assuming it is a yes-no question, the case in lines 2 to 6 would take effect. The *QuestionTypeYesNo* widget gets the *answer* object and an *onChangeYesNo(...)* function. The *QuestionManager* has a separate *onChange(...)* function for each input option. This gives the child widgets only the purpose of displaying the answers and the *QuestionManager* takes over controlling the answers. Listing 17, lines 8 et sqq. show the special question type where the interviewer decides whether to assign the disorder. If he assigns the disorder, *pushNextSection(...)* opens the next section (see 3.5.1).

```
1     switch (widget.question.questionType) {
2       case QuestionType.yesNo:
3         return QuestionTypeYesNo(
4           answer: answer,
5           onChangeYesNo: onChangeYesNo,
6         );
7       ...
8       case QuestionType.assignDisorder:
9         return QuestionTypeAssignDisorder(
10          answer: answer,
11          onChangeAssignDisorder: onChangeAssignDisorder,
12          pushNextSection: widget.pushNextSection!,
13          sectionID: widget.sectionID!,
14        );
15      ...
```

```
16    }
```

Listing 17: An excerpt of the *build()* function inside the *QuestionManager*. It uses a switch statement to determine the correct child widget depending on the *questionType*.

In Listing 18 two functions of the *QuestionManager* are shown. Line 1 et sqq. show the *onChange* function passed to the child widget. It is executed whenever the user enters an answer. It also triggers *upsertAnswer(...)*, which writes the entered answers to the database or deletes them from the database. The call of the function is connected with a previous evaluation (see Listing 18, line 3). This checks if the given answer is null, i.e., empty. This evaluated expression is passed as an argument to *upsertAnswer(...)* (see Listing 18, lines 6 et sqq.). If the expression evaluates to true, the answer will be deleted from the database (see lines 10-15). In addition, the current *answer* object must be reset. Otherwise, future database operations will be performed on an *answerID* that no longer exists. If the answer is not empty, it must be evaluated by prepending the *answerID* whether a new database entry is created or an existing entry is updated (see Listing 18, lines 15 et sqq.). If a new entry is created, the returned *answerID* by the database operation is stored in the current *answer* object. So that a database call is not triggered for every keystroke (e.g. for an input field for text), the calls just described are inside a *Timer* object. The timer runs for one second and only then executes the corresponding database operation. After the operations are called, the *updateInterviewPageContent()* function obtained from the parent widget is executed. This will then run the functions and adjust the interview content according to the answers (see Listing 14).
Another purpose of the *QuestionManager* is to hide or show questions if they are skipped due to applied jump rules. This behaviour depends on its argument *isVisible*, passed down by the parent widget *InterviewPage*. As soon as the variable changes, the appearance of *QuestionManager* changes accordingly. If *isVisible* is false, then the second part, i.e. the input options with the answers, is hidden. Additionally, only the question's first line is visible and gets a light grey, still readable font colour.

```
1   void onChangeYesNo(bool? value) {
2     answer.yesNo = value;
3     upsertAnswer(value == null);
4   }
5   ...
6   Future<void> upsertAnswer(bool delete) async {
7     ...
8     timeHandle = Timer(const Duration(seconds: 1), () async {
9       // entered answer is empty
10      if (delete) {
11        await AnswerModel.deleteAnswer(answer.answerID!);
12        // reset answer by deleting answerID
13        answer = Answer(widget.interviewID,
14                        widget.question.questionID);
```

```
15          } else {
16            // existing answer
17            if (answer.answerID != null) {
18              await AnswerModel.updateAnswer(answer);
19            }
20            // create new answer
21            else {
22              int newAnswerID =
23                        await AnswerModel.createAnswer(answer);
24              answer.answerID = newAnswerID;
25            }
26          }
27          widget.updateInterviewPageContent();
28        });
29    }
```

Listing 18: Two functions that show the interaction between user input and database calls.

3.5.4 DiagnosticToolHelper

The *DiagnosticToolHelper* has already been mentioned several times. This is not a widget but a pure helper class that computes various calculations. These are mainly calculations needed by the *InterviewPage*, for example, to decide whether certain questions should be displayed. Another large part takes care of the static jump rules (see 3.4.3), which differ from the dynamic jump rules in that they follow the same pattern for all sections (with a few exceptions). As mentioned in 3.4.3, one pattern represents one static jump rule. Each pattern is executed by one function. The execution of these functions is always done by *executeJumpRules()* inside *updateInterviewPageContent()* (see Listing 14, line 3). Listing 15 already indicates that the static jump rules are appended to the already executed jump rules (see Listing 15, lines 4 et sqq.). Each if block represents a static jump rule and executes one of the following four functions of *DiagnosticToolHelper*.

The first static skip rule checks whether at least one criterion question A is answered with yes. If this is the case, the criterion questions V are skipped. Listing 19 shows the simplified procedure of this function. The function gets a set of arguments: all criterion questions A and V, and the first question after the criterion questions, i.e. the first question of the present. First, a new object *jumpRule* is created with the properties from Listing 4. Here *sqlStatement* is empty, *startQuestionID* is the ID of the last question of the criterion question A, and *nextQuestionID* is the ID of the first question in the present. Then the individual criterion questions A and V answers are still retrieved from the database. Listing 19, lines 6 to 12 show the default case: once at least one criterion question A is satisfied, the function sets *queryResult* to true and returns the *jumpRule* object. If it is an exception case, all criterion questions A must

be satisfied (see Listing 19, lines 12-18).

```
1   static Future<JumpRule> executeStaticJumpRule_1(...) async {
2     List<bool?> resultCriterionPresent = [];
3     List<bool?> resultCriterionPast = [];
4     ...
5
6     if (!exceptions.contains(section.name)) {
7       // at least one criterionPresentQuestion: yesNo = 1
8       if (resultCriterionPresent.contains(true)) {
9         jumpRule.queryResult = true;
10        return jumpRule;
11      }
12    } else {
13      // for all criterionPresent: yesNo = 1
14      if (resultCriterionPresent.every((element) =>
15                element == true)) {
16        jumpRule.queryResult = true;
17        return jumpRule;
18      }
19    }
20
21    return jumpRule;
22  }
```

Listing 19: Presents the function that executes the first static jump rule.

The functions for the remaining static jump rules are very similar in structure and differ only in their conditions to be checked. The function of the second static jump rules receives the same arguments and checks whether all criterion questions A are answered with no and whether there is at least one criterion question V with yes. There are no possible exceptions to handle.
The third static jump rule is satisfied if all criterion questions A and V are answered with no. Here, instead of *nextQuestionID*, the variable *nextSectionID* is set. Usually, this is the next section in the sequence. However, some sections have well-defined successor sections. If the current section is in the list of exceptions, the ID stored in the list is used. As a result of a fulfilled third static jump rule, the current section is left and continues with the specified section.
The next case involves the criterion questions A and V and the question about the past. Therefore, the function for executing the fourth static jump rule receives an additional argument. The default case is that at least one criterion question A is answered with yes. The criterion questions V are skipped, so their answers are irrelevant. Instead, the answer to the question about the past is checked to see if it is answered no. If it is, it means that the disorder is not fulfilled in the present and the past does not want to be queried. Then the jump rule is fulfilled and the next section is opened. The exceptions are the same as for the first static jump rule. That means, for some

sections must be all criterion questions A must be fulfilled.

4 App Solution

Figure 4 shows a simplified version of the application's navigation tree focusing on the most important stages. Red circles represent the most significant pages or, rather, widgets. Dialogs, i.e. windows that do not cover the entire screen and have action buttons, are symbolized by blue circles. Drawers are orange, i.e., elements that slide in from the edge of the screen to provide navigation options or additional content to the user. At the same time, other less important pages are green. Based on Figure 4, the final implementation of the application is presented by showing screenshots of multiple pages to explain their functionality. The order of the pages corresponds to typical usage, starting with registration, moving on to the creation of patient documents and the entry of patient-specific data, followed by interviews and diagnosis.

4.1 Home

The very first visible screen is the splash screen when the user starts the application. It shows the FBZ logo next to the DIPS's name. When the application is ready, the home screen (see Figure 5) shows the FBZ logo and all users created on this device. Each user is represented by one card, including an icon and the username. Although, there is one card that has a different icon. This belongs to the admin. Cards create closed regions, suggesting their elements belong together (see 2.2.2). In addition, the shadows of the cards create a three-dimensional effect, which engages the user to click. Using the concept of knowledge in the head (see 2.2.7), a big plus sign on one card communicates, adding more users.
Figure 4 indicates three possible navigation paths. First, the imprint can be accessed by clicking the text button in the lower right corner. Second, clicking on a user card opens the login dialog. Moreover third, the registration dialog is opened by clicking on the plus sign.

4.1.1 Registration and Login

Clicking on the registration card opens the registration dialog (see Figure 6b). This dialog has three different text input fields for username, password and e-mail, where e-mail and password must be entered twice to avoid typos. The e-mail is important because it is needed to reset the password if it is forgotten. In addition to a label, the input fields have a leading icon to indicate their contents. For the text fields of the passwords, there is an option to display the input by clicking the trailing icon. If the user is an admin and there is no admin yet, a checkbox can be selected to create an admin instead of a user. The user is created after the input is complete and the button "Benutzer erstellen" is clicked. For this purpose, a password-protected connection to the database is established. The hashed password and all other relevant information

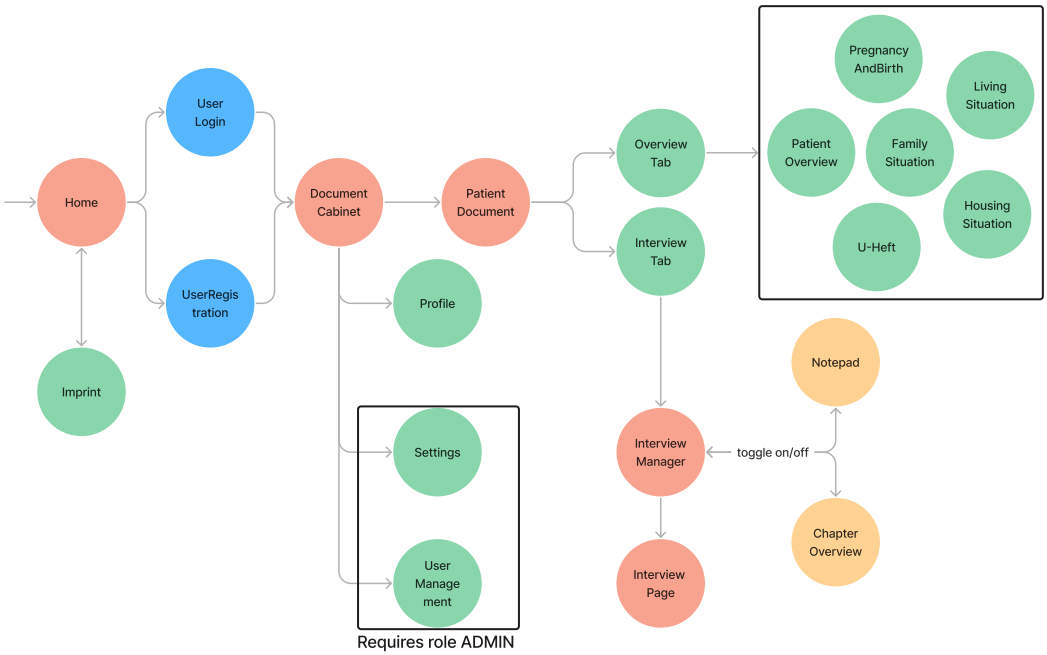


Figure 4: A simplified widget tree showcases the basic navigation. Red circles represent the most significant pages or, rather, widgets. Dialogs are represented by a blue circles. Drawers are orange circles. The basic pages are green.

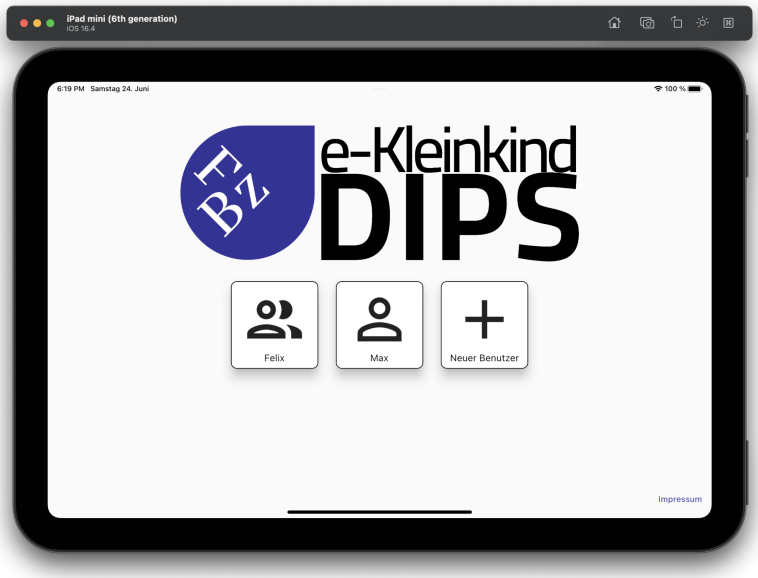
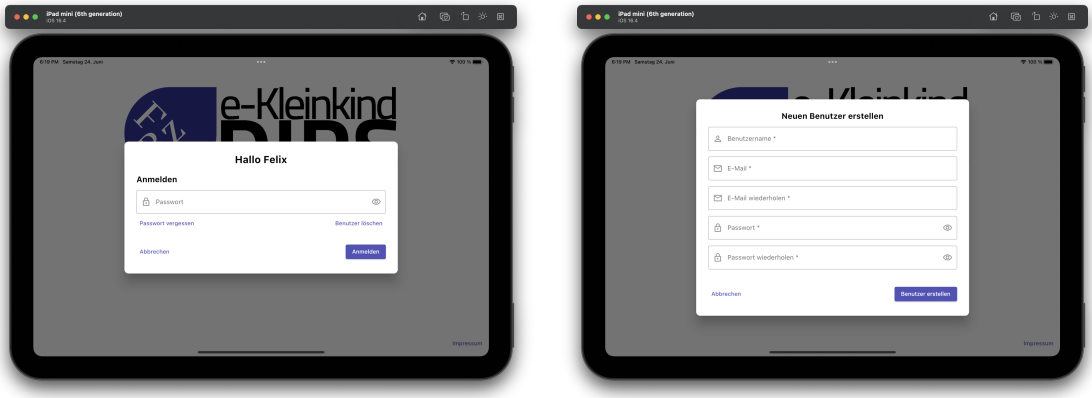


Figure 5: The first page after launching the app.



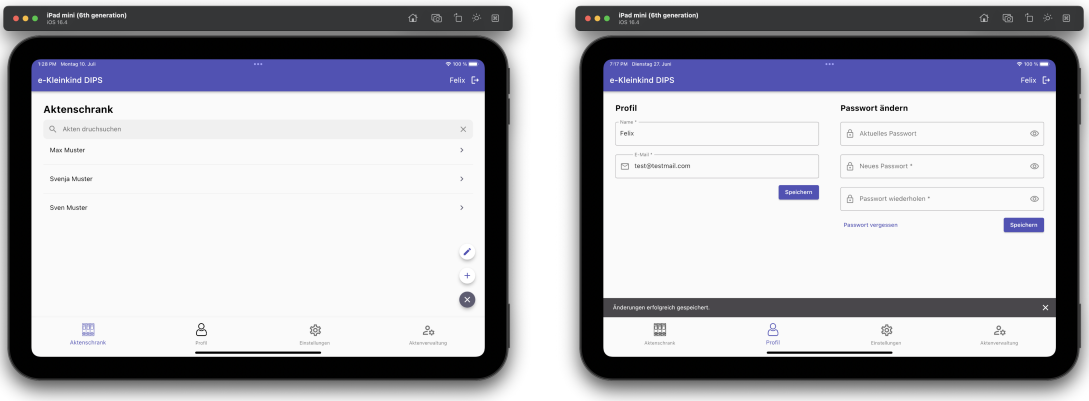
(a) (b)

Figure 6: (a) The login dialog to enter e-mail and password.
(b) If no account has been created yet, a new one can be created within the registration dialog.

are stored in the database. Finally, the user gets to his document cabinet. This is also the case if the user exists and has entered the correct password via the login dialog (see Figure 6a). Nevertheless, in addition, the login dialog offers two more functionalities. First, the user can request to reset his password by entering the correct e-mail associated with this user. The user will receive an e-mail with a code to enter for verification and can set a new password afterwards. The second function is to delete the user and all data attached to him.
Proper alignment creates visual balance and a professional look and feel. The consistent design of the login and registration dialog further strengthens this. (see 2.2.8) Additionally, informative feedback is provided when the password is incorrect, communicating the user to retry (see 2.2.6). The border of the input field is red and red information text is displayed below it.
As seen in Figure 4, both login and registration dialog arrive at the document cabinet after data is entered successfully.

4.2 Document Cabinet and Profile

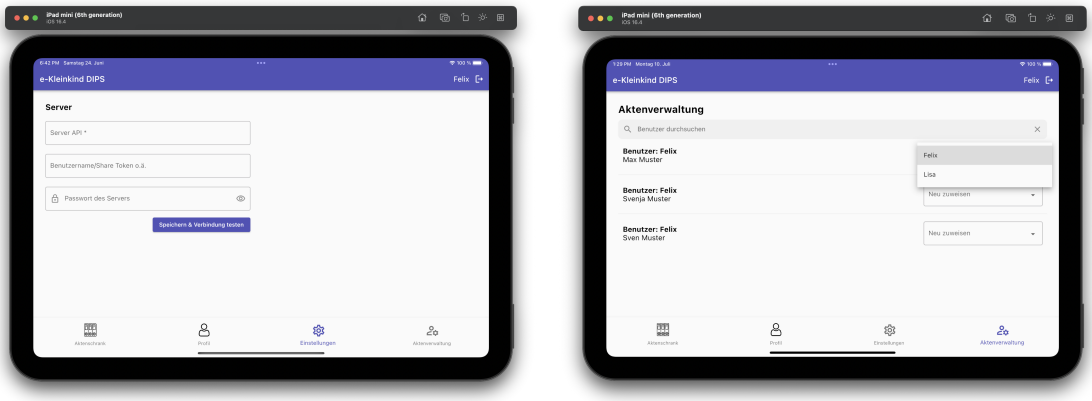
The document cabinet (see Figure 7a) lists all patient documents, or in case of a fresh account, a hint explaining how to create a new one. A text input field allows quick searching for specific patients. In the top app bar, the currently logged-in user and an icon for logging out are displayed at the right end. The bottom navigation bar offers two or four (as admin) icon buttons to navigate between pages. In the first place is the document cabinet. In the profile, personal data can be changed. The next two icons representing settings and document management are only visible to the admin. A floating action button with the burger menu icon indicates further actions. By clicking, the button expands into three smaller ones. The icons pen, plus, and cross



(a) (b)

Figure 7: (a) The document cabinet shows all created patient documents.
(b) In the profile, personal data can be edited.

represent the actions edit, add and close. By using the concept of knowledge in the head, no further labels are needed keeping the design clear and simple (see 2.2.7). In the edit mode, patient documents can be deleted or archived. Therefore, a trash can and an archive icon replace the list tile icon removing the functionality to navigate to the patient document by clicking the entry. This prevents accidental navigation to the patient file, although it should be deleted or archived. In addition, the icons have precise colors. The trash can icon is red and uses the principle of cultural constraints (see 2.2.5), which makes users immediately suspect a *dangerous* action. The action must be confirmed in a subsequent dialog by clicking a button to prevent the user from unintentionally deleting this file. Similarly, the icon for archiving has a green colour, suggesting a *non-dangerous* and reversible action. The plus icon allows creating new patient documents and the cross icon closes the floating action button and changes into the initial slightly larger button with the burger menu icon.
The profile page (see Figure 7b) can be accessed without being an admin. Here user can change his personal data. The page is divided into two areas. By providing slightly more space between the areas than between the elements within them, they are perceived as a group due to the Gestalt law of proximity (see 2.2.2). The left side concerns the username and the e-mail. Moreover, on the right, the password can be changed. The same elements ensure a consistent design, as in the registration or login dialog. Additionally, the steps to reset the password are also the same. If the user clicks the save button, a progress indicator replaces the button until the system call is executed. The system provides feedback by showing a snackbar with an appropriate message at the bottom.



(a) (b)

Figure 8: (a) This view is only visible to the admin and is responsible for setting up a server.
(b) This view is also only visible to the admin, who can reassign patient documents to another user.

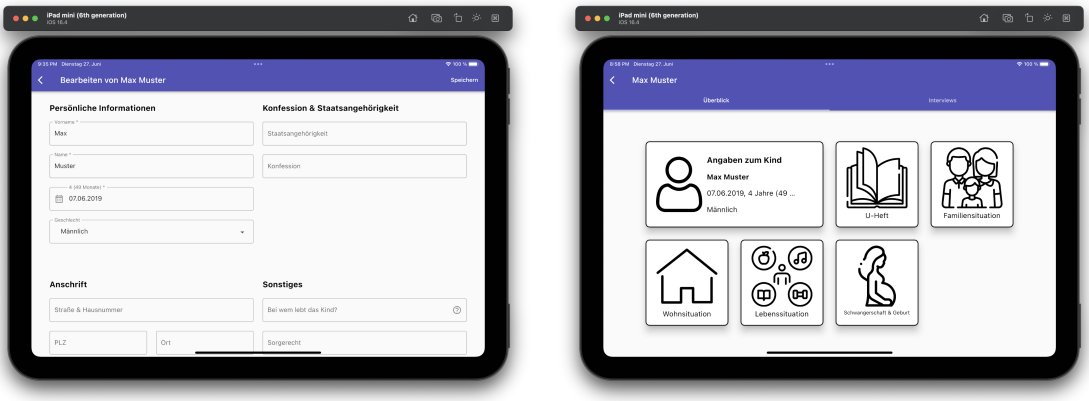
4.2.1 Settings and Document Management

As presented in Figure 4 by the black-bordered rectangle, the settings and document management view are only accessible by the admin. The admin can set the required server information on the settings page (see Figure 8a) to enable all users using this device to export their interview data. After entering the server data, the admin can test the connection. Again, the button is replaced by a progress indicator until the action is completed. Then a snackbar with a matching message appears. So far, only export to an ownCloud server[16] has been tested.
There may be times when an interviewer leaves the practice. To ensure the data is not lost, the admin can view and manage all data by reassigning patient documents. As shown in Figure 8b, the administrator can access all patient documents and view their owners. In order to reassign a patient document, he can select a new user by using the dropdown menu at the corresponding entry.

4.3 Patient Document

If the user selects an entry, e.g. Max Muster, in the document cabinet (see Figure 7a), the user is navigated to the overview of the patient document (see Figure 9b). This page is divided into two parts and contains several functionalities. In the first view, the user has an overview of the patient data sorted by categories represented by corresponding cards, as presented in Figure 4 by the black-bordered rectangle containing six different green circles. In the second view, the user can access interview-related information like creating new interviews or viewing existing ones (see Figure 10b).

[16]https://owncloud.com/

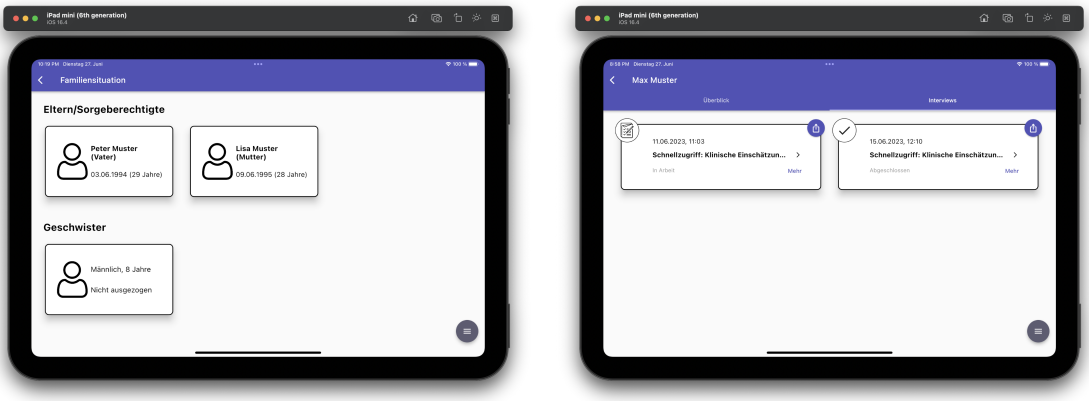


(a) (b)

Figure 9: (a) On this page, the user can create or edit a patient document.
(b) This view shows all patient-related data ordered by categories.

4.3.1 Overview Tab

The user is on the overview tab and can access to all patient-relevant data (see Figure 9b). If the user wants to change information about the patient, this is possible via the first slightly larger card (see Figure 9a). Here, optional data can be entered in addition to the mandatory data. A title and alignment of the different input fields create individual groupings for the different contexts (see 2.2.8). In addition, the individual groups are placed below each other if a smaller display is used, such as on a mobile phone. When data has been changed, but the user clicks on the back arrow at the top left of the header, a dialog warns of data loss. Either the user confirms the action and his changes are lost and returns to the previous page, or he cancels the action and can continue editing or saving the data by clicking the text button at the top right of the header. This input mask is also displayed when a new patient is created via the plus icon button (see Figure 7a). The user enters the patient data overview only after entering the mandatory data.
The information from the examination booklet is also relevant to the interview. It is possible to take photos of the examination booklet to save time. For this, the user clicks on the card *U-Heft* and is taken to a photo gallery. There is an expandable floating action button. When expanded, the first one shows a pencil icon and activates the edit mode, similar to the *Document Cabinet* in 4.2. In edit mode, the photos get a trash can icon in the top right corner and can be deleted. The captured photos can be viewed in enlarged versions by clicking on them in the gallery. The camera icon activates the photo mode, where the user can take photos of the U-book. With regard to Fitts' Law, the floating action button is quickly accessible because its size is appropriate for touchscreens and it is easy to reach because there is enough distance from other interaction elements (see 2.2.1). If the user selects *Familiensituation*, he will get to the view from Figure 10a. In addition to information about the patient, i.e.



(a) (b)

Figure 10: (a) If the patient has siblings, the user can add, remove, or edit this information on this page.
(b) Each interview created is listed here.

the infant, information about the family is also important. For this, on the one hand, reference persons such as parents or grandparents can be entered. For this purpose, there are two plus icons in the expandable floating action button. Each icon has a signifier to clarify which plus is for custodians or siblings (see 2.2.3). After selecting the plus icon, a page for entering the information appears. Finally, each custodian is represented by a card. If changes need to be made to the people, the card can be clicked. Afterwards, the user is taken to the same page when creating the person, with the difference that the data of the selected person is pre-filled in the corresponding fields. The pencil icon activates the editing mode and behaves in the same way as the gallery in the examination booklet.
As for the already explained cards, information corresponding to the category can be entered in the remaining cards.

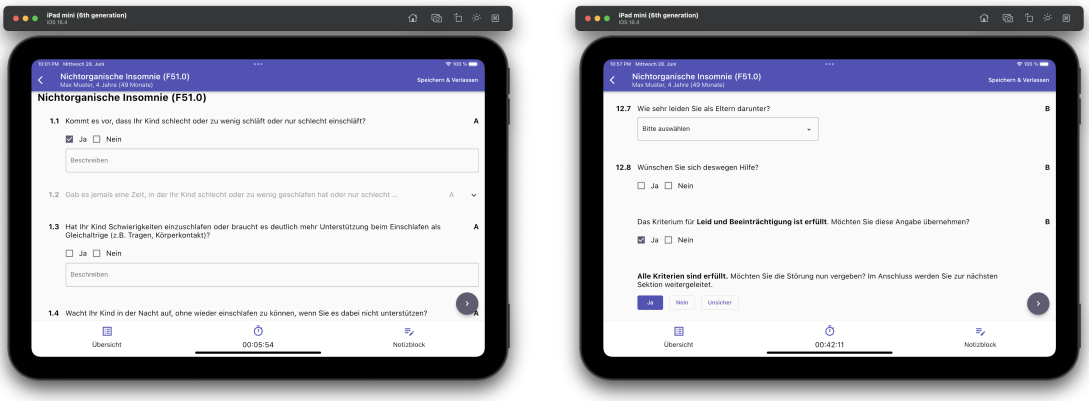
4.3.2 Interview Tab

As indicated in Figure 4, the user can switch the tab on the patient document view between overview and interview. In Figure 10b the interview tab is shown. Just like the other pages, there is an expandable floating action button. The pen activates the edit mode. The plus opens a dialog and starts a new interview. The info icon opens a bottom sheet, i.e., a window extending over the whole screen width and about half the screen height. Here, information on how to conduct the interview is described.
For each interview that exists for this patient, there is a card. A Card indicates the status by an icon in the upper left corner and text in the last line in the lower left corner. In Figure 10b, there is an interview in progress on the left and an interview completed on the right. An Export icon button is at the right corner of the card. This

exports the corresponding interview and the patient data from the overview. When the export starts, the icon is replaced by a progress indicator until the action is completed. Afterwards, a snackbar is displayed on the bottom, indicating either the success or failure of the action. The export is only available if the admin has already created a server on this device. Exporting the interview data creates a CSV file containing all information about the patient and the interview. A timestamp shows the creation date in the interview card's first line. Below that, there is quick access to the clinical assessment and diagnosis. At the right end of the last line, the text button *Mehr* opens the same dialog box as when creating a new interview, but in edit mode. The meta information entered when creating the interview can be edited and saved.
When a new interview is created via the plus icon, meta information can be entered within a dialog. The interviewer's name is mandatory and pre-filled with the name of the logged-in user. All other information is optional, so an interview can be started without user input and only with a click on *Start*.

4.4 Interview and assign Disorder

Once a new interview is created, the user is in the *InterviewManger* (see Figure 4). In Figure 11a, there is already an advanced section *Nichtorganische Insomnie*. The section's title and additional patient information can be seen in the header. Furthermore, if the dialog for creating the interview is open, the custodians are also displayed here. On the far right, is it possible to exit the interview anytime and return to the interview tab. The page content shows the individual questions (see 3.5.2, 3.5.3). For example, yes-no questions are shown, which additionally allows a description. Figure 11a shows how the interview content behaves when a static jump rule (see 3.4.3) is satisfied. In this case, questions 1.1 and 1.2 are marked as criterion question present and past, respectively. The user can only tell that they are criterion questions by the bold numbering and letter. The markings for criterion question A and V are only used for internal calculations. Since at least one (or in this case all) criterion question present is fulfilled, i.e. answered with yes, a static jump rule is applied and the criterion questions past are greyed out. The user thus recognizes that question 1.2 is no longer relevant and continues with question 1.3.
The user continues the interview and finally reaches the end of the section (see Figure 11b). In this case, all criterion questions have been answered with yes and in addition, suffering and impairment have been fulfilled and explicitly assigned by the user (see the second last question in Figure 11b). The question for assigning the disorder is then displayed. Initially, all three buttons are white and unchecked. The user then selects whether the disruption should be assigned, unassigned, or marked as not sure. This functionality was created because users must be explicitly asked to decide on assigning the disorder. This originates from the fact that the application is not (yet) a medical product, and therefore it is not allowed to make an independent diagnosis. The user is then redirected to the next section.
Apart from this situation, the user may want to change the section independently. The floating action button with the arrow pointing to the right is used for this purpose.



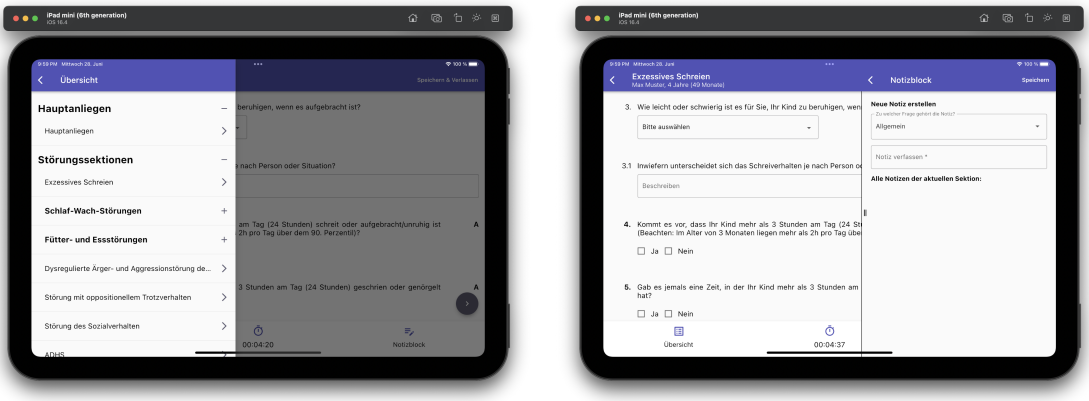
(a) (b)

Figure 11: (a) An interview section where a jump rule is applied to hide an unnecessary question.
(b) If the disorder is fulfilled, the user must decide whether to assign the disorder.

If this is clicked, the next section opens. The concept of perceptual mappings (see 2.2.4) helps here. The user clicks the button (action) and perceives the section change (effect).

4.4.1 Chapter Overview and Notepad

Figure 4 shows that the *InterviewManager* has two drawers; one is the *ChapterOverview* and the other is the *Notepad*. Both drawers can be opened and closed at any time during the interview. The *ChapterOverview* can be controlled via the left icon button. An overview of the sections opens from the left (see Figure 12a). These are divided into chapters and subchapters for a better overview. Furthermore, all chapters and subchapters can be expanded and collapsed by the corresponding plus or minus. If a section is selected, the application navigates to the corresponding section (see *pushNextSection(...)* in 3.5.1). The icon button on the bottom bat at the right opens the *Notepad*. Experiences with the paper-based version of the interview show that many notes are written on the sheets during the interview. Because there is no collected place for all notes, chaos quickly occurs. The implemented *Notepad* is supposed to help by offering to attach a note to a question or section. Figure 12b shows a dropdown to select whether it is a general note or a note linked to a specific question. Below is an input field to fill the note with the actual content. Once the data is filled in, clicking on the top right saves the note. In the lower half of the *Notepad* all notes of the section are listed. The title will then contain either the corresponding question number or *Allgemein* if it is a general note of the section. The edit mode is activated here by selecting a note. It will be highlighted with the primary colour and a trash can icon will appear in the upper right corner. In addition, the input fields are filled with the



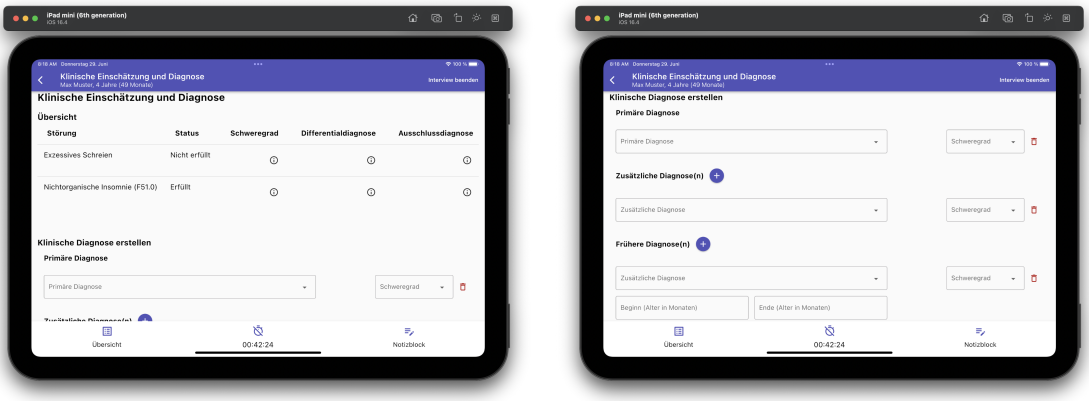
(a) (b)

Figure 12: (a) The chapter overview helps with fast navigation.
(b) Notes can be written down to a specific question or the current section.

data of the selected note. The user can edit the content of the selected note. Changes are then applied with *Save*. Selecting the same note again deactivates the edit mode. Compared to *ChapterOverview*, the open *Notepad* does not obscure the rest of the interview. This is intentional and is to allow for writing down the note and reading the interview content at the same time. Also, the drawer can be resized by dragging the left border according to the desired size.
In the middle of the bottom bar is a stopwatch, which automatically records the time of the interview. The user can stop and restart the time by clicking the stopwatch icon.

4.4.2 Clinical Assessment

After the interview, the interviewer must determine the clinical diagnosis. As soon as the user gets to the clinical evaluation and assessment page, the stopwatch stops automatically. First, the user sees the list of (processed) disorders (see Figure 13a). It shows the disorders and their markings, i.e. whether the disorder has been assigned, unassigned (*Nicht erfüllt*) or marked as unsafe. The horizontally scrollable table also displays other information. The third column contains the severity of the disorder calculated by the application or a dialog for determining the severity. Next to it, dialogs open in the two other columns containing information about the differential and exclusion diagnoses. This information is different for each section.
Below the disorder overview is the diagnosis input mask (see Figure 13b). The clinical diagnosis is divided into primary, additional and previous diagnoses. Only name and severity are needed for the primary diagnosis. There are predefined names of possible disorders. However, the option *Andere* allows the user to input an individual name into a new text input field. A number from 0 to 8 has to be selected for the severity level. Here the concept of mapping has been applied. The numbers are arranged from



(a) (b)

Figure 13: (a) A list of disorders helps the user to make a clinical diagnosis.
(b) This excerpt shows the input mask to enter the clinical diagnosis.

top to bottom, from 8 to 0. Thus, 8, since it is at the top, is mapped with a lot and 0, since it is at the bottom, is mapped with a little (see 2.2.4). A plus icon is next to the subtitle of the tables for the additional diagnoses as well as previous diagnoses. With a click, another line is added accordingly. Next to each entry, the entry can be deleted by clicking on the trash can icon. If it is the last entry, only the content is deleted, not the complete entry. For the earlier diagnoses, in addition to the name and severity, there are two more entries for the age at the start and end time.
Once the user completes with the clinical diagnosis and assessment, the user can click on the *Interview beenden* text button in the upper right corner. Consequently, the interview is closed and its status changes from in progress to done. Back on the interview list in the interview tab (see 4.3.2), the icon changes to a checkmark and the user has successfully completed the interview.

5 Comparison

The following pages describe the setup of the user study and its evaluation. First, it is described how the user study is built, followed by a description of its execution. Then, the results are divided into two parts and analyzed. The results of the first part refer to the System Usability Scale questionnaires. The second part compares the actual value of the app-based interviews and the target values of the comparison interview.

5.1 Setup

The user study tests the two different systems, the paper-based interview guideline and the newly developed application. Fourteen testers participated in the user study. They are students from postgraduate training to become a psychotherapist and most

of them were already trained in the paper-based version of a DIPS for older children. For the user study, an interview between the interviewer and patient was simulated and videotaped. A female psychologist simulated the mother of the five-year-old patient suffering from sleeping problems and anxiety. The symptoms were scripted before the interview, adapted from an actual interview. The interviewer did not know the content beforehand.
The user study follows the Between-Groups design. This means that each tester tests both systems. On the one hand, this requires fewer testers and, on the other hand, reduces individual differences between the testers. However, a certain learning effect can occur between the execution of the test using different systems. This effect can be counteracted by ensuring that the order of the used system differs from tester to tester. [1]
After using one system, testers complete the System Usability Scale (SUS) questionnaire. With the help of this questionnaire, a UI can easily be evaluated in terms of usability. The questions are standardized and can be answered using a Likert scale. The Likert scale consists of five answer options, with a value of five corresponding to *I completely agree* and a value of 1 corresponding to *I completely disagree*. A user interface can score between 0 and 100 points in the evaluation. Values of 85 and more describe excellent usability. In contrast, systems with values of less than 70 can only be used poorly. The SUS questionnaire is a recognized tool for quick and rough assessment of systems. [7]
In addition, the testers using the mobile application exported their interview data. These exported CSV files are compared with the target values of the test interview.

5.2 Execution

First, testers receive an introduction to the use of DIPS in general. Then they are introduced to the paper-based interview guideline and the app. After the theoretical part, the testers are divided into two groups since all testers watch the recorded test interview together. The first group uses the paper-based interview guideline and the second group uses the application on iPads while watching the test interview the first time. Subsequently, the testers of both groups completed the SUS questionnaire. Afterwards, the test interview is watched again, but the groups exchanged their systems. This means that now the first group uses the app on the iPads and the second group uses the paper-based interview guideline.

5.3 Evaluation System Usability Scale

Since the recorded test interview has a fixed length, there is no comparison between the systems in terms of their duration of use. Instead, the focus of this comparison is on the degree of usability. Figure 14 presents the results of the SUS questionnaire by calculating the average values per question of all testers per system. On the left are the results from the app and on the right are those from the paper-based interview guideline. Bar chart values range from 1 to 5, with 1 being the best value for even

(grey) questions and 5 being the best value for odd (blue) questions.
Testers would like to use the paper-based interview guideline less often than the app. With a value of 4.3 out of 5, the app is 0.9 points higher in reuse. The testers' comments reflect this. Among other things, the paper-based interview guideline was perceived as more confusing and harder to understand. This is also demonstrated in the next question about how (unnecessarily) complex the system is perceived to be. The app scored very well at 1.4, whereas the paper-based system was rated more complicated on average at 2.5. The next question asks whether the system was easy to use. The app has an almost full score of 4.1. Nevertheless, the paper-based interview guideline also scored well at 3.4. Still, there is a clear trend toward the app. The fourth question is about 2 for both systems and shows that both systems are usable without help. Next, the 5th question addresses the integration and coherence of the system's functions. Testers perceived the app's various functions as seamlessly connected and working together effectively (4.2). For the paper-based system, the value is exactly 3. Consequently, the integration of the functions appears less well. Next, design, terminology, and functionality are rated for inconsistency. Here, the app is consistent at 1.5. The other system is also consistent overall at 2.2. Question 7 assesses the learnability of the system, showing that learning the app (4.3) is easier than learning the paper-based interview guideline (3.3). Using the app (1.6) was more effortless than using the paper-based system (2.5). For the last two questions, the scores for both systems are nearly the same. Question 9 asks how confident users feel about using the system. In both systems (app 3.2, paper-based system 3.4), the feeling of security is in the middle range. This score is probably due to the inherent complexity of DIPS. The 10th question assesses how much effort needs to be invested in understanding and using the system effectively and fully. The ratings are similar for both the app (2.3) and the paper-based interview guideline (2.5). The amount of learning required to use the system effectively seems low for both systems.
In addition to comparing the values for each question, an overall score can be calculated for the respective system. For this, a user's input is normalized by subtracting the input value from 5 for even questions and decreasing the input value by one for odd questions. These values are summed, resulting in the SUS score. The SUS score of a system is the average of all SUS scores of that system. For the app, this results in an average SUS score of 78.3. A score of 85 to 100 represents excellent usability and a score below 70 represents poor usability [7]. So the testers perceive the app as good in terms of usability, but it has individual areas for improvement. The paper-based interview guideline achieved an average SUS score of 61.4. Thus, the threshold value of 70 is not reached. Consequently, the perceived usability is significantly worse.

5.4 Evaluation of Actual and Target Values

There is a completed paper-based interview guideline of the recorded test interview, the entries of which are regarded as target values. The exported interview data of the application correspond to the actual values. Certain conclusions can be drawn based

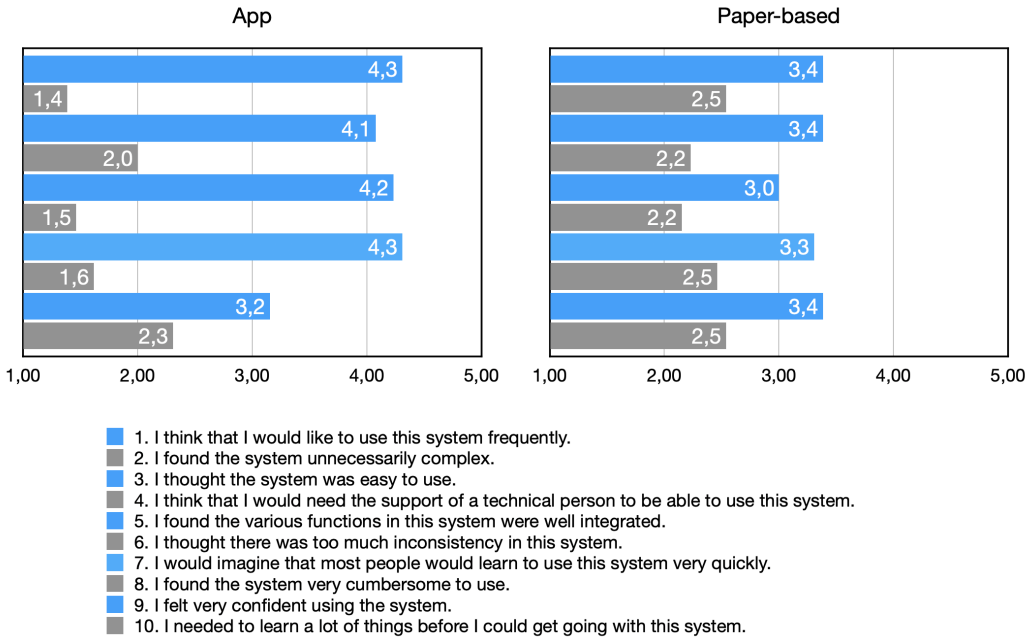


Figure 14: Results of the System Usability Scale questionnaire. On the left the results of the app and on the right, the results of the paper-based interview guideline are shown. For even (grey) questions is 1 the best value and 5 is the best value for odd (blue) questions.

on the exported data.
The exported answers almost match the target values at the section level. Some small differences exist between testers. For example, only the corresponding criterion questions A and V are answered regarding the skipped sections. As a result, the section is left and the next one is opened. This means that this kind of jump rule works. Furthermore, it can be seen that the other static jump rules also work correctly. In addition to the static one, the dynamic jump rules, e.g. to hide follow-up questions, can be validated properly.

When evaluating the diagnoses of the interviews, everything seems right at first glance. After all, the app users had all made the correct diagnosis compared to the comparison interview. However, the exported data show no disorders (see 4.4.2 and Figure 13a). The data suggest that the fulfilled sections still needed to be completed. There is no justified reason for this. It is reasonable to assume that the questions about assigning the disorder and the question about suffering and impairment were not seen. Consequently, the system cannot decide whether the disorder is fulfilled.
In order to evaluate whether the system displays the questions at all, the answers of the two sections were entered manually into the application for each exported data set. It was found that both questions were displayed. When these two questions were subsequently answered appropriately, the list of disorders at the end of the interview also indicated the correct disorders.

6 Conclusion

The primary objective of this research was to investigate the feasibility of replacing traditional paper-based diagnostic interviews with an Android/iOS application. The study aimed to assess whether an app can effectively replicate the functionalities of paper-based interviews while offering additional advantages in accessibility, efficiency, and accuracy. By comparing the app's performance with the conventional paper-based interview method, this thesis evaluated its potential to simplify the interview process, improve data collection and analysis (by exporting to a statistics-friendly format), and enhance the overall quality of diagnostic assessments. Through the evaluation and feedback of the user study, the thesis aimed to provide valuable insights into the viability and usability of app-based diagnostic interviews in the context of therapeutic practices.

The key findings of this research demonstrate that the mobile application to conduct structured interviews shows promising potential as a replacement for the traditional paper-based method. The testers reported that the app was useful in facilitating the interview process. The system usability scale (SUS) score further supported these observations, with the app receiving a higher score of 78.3 compared to the paper-based interview guideline's score of 61.4.

Although the system did not display the two entries in the list of disorders in the user study, it was still possible to make the same diagnosis as the comparison interview. In addition, a review of the exported data revealed that the entries in the list of disorders would be present if the two initially hidden questions were completed. Overall, the clinical assessment conducted through the app yielded comparable results.

It is worth noting that some users expressed uncertainty while conducting the interview, possibly due to the unfamiliarity and perceived difficulty of DIPS. However, when the app was used, the interview was perceived as less difficult and a slight reduction in uncertainty was observed.

These findings highlight the viability of replacing the paper-based interview guideline with an accurate app.

6.1 Future Work

The most crucial area that needs to be adjusted is the visibility of the initially invisible questions that appear under certain conditions. The user study clearly shows that the questions were not seen, so the system assigned no disorders. An appropriate solution is needed so users are aware of these questions. There are several possible methods to improve this: On the one hand, a snackbar could be used to inform the user that something has happened further down the page. On the other hand, it is also possible to scroll automatically to the new question. However, this would be highly immersive and, therefore, might negatively impact usability. Perhaps a combination is a better solution, in which an icon button acts as a hint and, with a click, then scrolls to the desired content. Additionally, it would be possible to shift the decision from being questions to dialog windows. These will catch the user's attention. Furthermore, it

can be set that these dialogs cannot be clicked away, but a user action is required. This can ensure that the user makes a decision. These ideas need to be further evaluated by users of the target group.
As introduced in 3.1, the diagnostic criteria are based on the DSM-5. However, there are two more important classification manuals: ICD-10 and DC:0-5. It is helpful if the diagnoses can be sorted according to the different classification manuals. Some criterion questions vary in content between the manuals or are not criteria at all.
The system can also be further developed regarding of diagnosis and its severity. It is possible to automatically determine the severity of a few disorders based on answers to certain questions. It is worth validating whether the remaining sections' severity can also be determined automatically, perhaps by intentionally involving severity questions.
In addition to DIPS for infants (Kleinkind-DIPS), there are also the Kinder-DIPS (6-18 years), Mini-DIPS (from 16 years) and DIPS (from 16 years) [14]. Since maintainability, scalability, and reusability were all considered during the whole development process, the app may be expanded with these additional DIPS without damaging any already-existing components and instead probably benefit from these components. This is because the other DIPS differ from the Kleinkind-DIPS mainly in terms of content. There are many ways to integrate another DIPS into the app. It is only necessary to place another instance in a suitable place, making it possible to choose between the DIPS. For example, this can be already on the home page before the user logs in. The user would first decide which system he wants to log into through a selection menu. After logging in, he can then always use only the corresponding DIPS. The advantage is that the content would be separated according to the DIPS. On the other hand, switching between the systems would always mean logging out and logging in again. Additionally, there would be one patient record list per DIPS. Option two would expand the current system to offer a choice of available DIPS when the interview is created. The user then selects the desired DIPS. Thus, there would be only one list of patient records. Then it would have to be evaluated whether marking the patient files would be necessary so that it would be recognized faster to which DIPS the respective entry belongs. Switching between the possible DIPS would be easier, as there would be no need to log out and log in.
Many other possible ideas can be implemented. In any case, the users of the target group should be sufficiently involved to meet their needs in the best possible way.

References

[1] Bernard C Beins. Within-groups design. *The Encyclopedia of Cross-Cultural Psychology*, 3:1355–1356, 2013. doi: https://doi.org/10.1002/9781118339893.wbeccp570.

[2] Chunyue Bi. Research and application of sqlite embedded database technology. *WSEAS Trans. Comput*, 8(1):83–92, 2009. URL `http://wseas.us/e-library/transactions/computers/2009/31-846.pdf`.

[3] Katrin Bruchmüller, Jürgen Margraf, Andrea Suppiger, and Silvia Schneider. Popular or unpopular? therapists' use of structured interviews and their estimation of patient acceptance. *Behavior therapy*, 42(4):634–643, 2011. doi: https://doi.org/10.1016/j.beth.2011.02.003.

[4] S. Dow, B. MacIntyre, J. Lee, C. Oezbek, J.D. Bolter, and M. Gandy. Wizard of oz support throughout an iterative design process. *IEEE Pervasive Computing*, 4 (4):18–26, 2005. doi: https://doi.org/10.1109/MPRV.2005.93.

[5] John D. Gould and Clayton Lewis. Designing for usability: Key principles and what designers think. *Commun. ACM*, 28(3):300–311, mar 1985. ISSN 0001-0782. doi: https://doi.org/10.1145/3166.3170.

[6] Lisa Graham. Gestalt theory in interactive media design. *Journal of Humanities & Social Sciences*, 2(1), 2008. doi: https://doi.org/10.33140/JHSS.

[7] Rebecca A Grier, Aaron Bangor, Philip Kortum, and S Camille Peres. The system usability scale: Beyond standard usability testing. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 57, pages 187–191. SAGE Publications Sage CA: Los Angeles, CA, 2013. doi: https://doi.org/10.1177/1541931213571042.

[8] ROGER R HALL. Prototyping for usability of new technology. *International Journal of Human-Computer Studies*, 55(4):485–501, 2001. ISSN 1071-5819. doi: https://doi.org/10.1006/ijhc.2001.0478.

[9] Andreas Holzinger and Stephen Brown. Low cost prototyping: part 2, or how to apply the thinking-aloud method efficiently. In *Proceedings of the 22nd British HCI Group Annual Conference on HCI 2008: People and Computers XXII: Culture, Creativity, Interaction - Volume 2, BCS HCI 2008, Liverpool, United Kingdom, 1-5 September 2008*, pages 217–218, 01 2008. doi: http://dx.doi.org/10.1145/1531826.1531897.

[10] Nishtha Jatana, Sahil Puri, Mehak Ahuja, Ishita Kathuria, and Dishant Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research & Technology*, 1(6):1–5, 2012.

[11] Thomas E Joiner Jr, Rheeda L Walker, Jeremy W Pettit, Marisol Perez, and Kelly C Cukrowicz. Evidence-based assessment of depression in adults. *Psychological assessment*, 17(3):267, 2005. doi: https://psycnet.apa.org/doi/10.1037/1040-3590.17.3.267.

[12] Janin Koch and Antti Oulasvirta. Computational layout perception using gestalt laws. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1423–1429, 2016. doi: https://doi.org/10.2214/AJR.07.3268.

[13] I. Scott MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Human–Computer Interaction*, 7(1):91–139, 1992. doi: https://doi.org/10.1207/s15327051hci0701_3.

[14] Jürgen Margraf, Jan Christopher Cwik, Verena Pflug, and Silvia Schneider. Strukturierte klinische interviews zur erfassung psychischer störungen über die lebensspanne. *Zeitschrift für Klinische Psychologie und Psychotherapie*, 2017. doi: https://doi.org/10.1026/1616-3443/a000430.

[15] Tina Matuschek, Sonia Jaeger, Stephanie Stadelmann, Katrin Dölling, Steffi Weis, Kai Von Klitzing, Madlen Grunewald, Andreas Hiemisch, and Mirko Döhnert. The acceptance of the k-sads-pl–potential predictors for the overall satisfaction of parents and interviewers. *International Journal of Methods in Psychiatric Research*, 24(3):226–234, 2015. doi: https://doi.org/10.1002/mpr.1472.

[16] Murielle Neuschwander, Tina In-Albon, Andrea H Meyer, and Silvia Schneider. Acceptance of a structured diagnostic interview in children, parents, and interviewers. *International journal of methods in psychiatric research*, 26(3):e1573, 2017. doi: https://doi.org/10.1002/mpr.1573.

[17] Donald A. Norman. *The design of everyday things*. Basic Books, [New York], 2002. ISBN 0465067107 9780465067107.

[18] Jean M Twenge and Thomas E Joiner. Mental distress among us adults during the covid-19 pandemic. *Journal of Clinical Psychology*, 76(12):2170–2182, 2020. doi: https://doi.org/10.1002/jclp.23064.

[19] Robert A. Virzi, Jeffrey L. Sokolov, and Demetrios Karis. Usability problem identification using both low- and high-fidelity prototypes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '96, page 236–243, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917774. doi: https://doi.org/10.1145/238386.238516.

[20] Robin Williams. *The non-designer's design book: Design and typographic principles for the visual novice*. Pearson Education, 2015. ISBN 9780321193858.