ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Databases and Information Systems Department

Conception and Realization of a Chatbot-System to support Psychological and Medical Procedures

Master's thesis at Universität Ulm

Submitted by:
Jens Winkler
jens.winkler@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert
Dr. Rüdiger Pryss

Supervisor:
Robin Kraft

2019

Version from January 23, 2019

Composition: PDF-LaTeX 2_ε

Abstract

As a result of long term researches of Artificial Intelligent, the influence as well as the amount of possible use-cases of chatbots is growing constantly. As to health care, chatbots can be used to simplify the interaction between experts and patients. For example, chatbots can help people who are affected by depression, as the barrier to chat with an application is less high than meeting a real person. While time flexibility can be a benefit for patients, the response to frequently asked questions relieves experts so they can focus on important issues. Consequently, the usage of chatbots in psychological and medical sectors can be a step to improve the health care system.

The objective of this master's thesis is to create a system concept involving a conversational agent, a mobile application and a back-end application to built solutions for the psychological and medical sectors. As all possible use-cases can not be clearly restricted, the concept is designed to be adjustable. Referring to health care, the systems needs to be able to react to critical situations like medical emergencies. However, handling life-threatening cases exceeds the capability of the system and always needs human assistance. Therefore, the concept also includes human participants such as experts. To test the approach, a mobile application was developed that includes a simple conversation using the AI assistant *IBM Watson Assistant*. In addition, a generic framework was developed to handle multiple third-party chatbots. Furthermore, the mobile application reacts to critical situations, such as the detection of suicidal thoughts or if the user enters an input that is unknown for the system. Moreover, the knowledge base of the chatbot can be extended using a conversation formatted process.

Acknowledgment

I would like to thank everyone who supported me during this master's thesis.

Very special thanks to Dr. Rüdiger Pryss and Robin Kraft for the support and the assistance during my master's thesis.

I would also like to thank Eileen Bendig for providing me with a psychological interview that is used as the content of the knowledge base of the chatbot.

Furthermore, I thank my proofreaders for their help to improve the writing and wording of this master's thesis.

Contents

1

Introduction

The influence and the spreading of chatbots in the society grows constantly. The size of the worldwide chatbot market in 2016 was said to be worth 190.8 million U.S. dollars. In 2025, the value of this market is assessed to 1250 million U.S. dollars [1]. This indicates an increase of about 600 percent. The expansion of this sector can be a solution of the insufficient quality of health care in certain countries. According to *Ipsos*, 45 percent of adults worldwide rated the quality of the health care in their country as good. 37 percent said it was neither good nor poor and the remaining 23 percent rated the health care they had access to as poor. While *Great Britain* scores well with 73 percent, *Germany* reaches 56 percent and Poland only 14 percent [2].

Possible reasons for the dissatisfaction can be the long duration of getting a general practitioner (GP) or a hospital appointment. Also the personell shortage in todays' hospitals can be a possible cause for the bad ratings. Individuals of the *United Kingdom* (UK) rated these two aspects with 52 percent and consequently as the highest reason of their dissatisfaction with the National Health Service (NHS). Furthermore, 22 percent of the study participants were not happy with the quality of provided NHS services [3]. Besides, german inhabitants criticize that doctors do not have enough time for their patients or that they are unhappy with the opening hours of medical surgeries. Furthermore, they felt that they are not taken seriously by medical specialists [4].

The usage of chatbots could reduce the mentioned discontent. As machines do not need a break like human beings, they could provide a 24 hour service for everyone who has access to a smartphone or to a computer. Patients therefore do not need to conform with the opening hours of medical facilities. Furthermore, chatbots are able to reply instantly and avoid that patients need to wait for hours until they can meet a

medical specialist. The instant response of a chatbot is also expected by 55 percent of customers of the *United States* [5]. Due to the fact that the majority of our society has access to a smartphone, virtual assistants can also increase the accessibility of medical or psychological support. Inhabitants of small towns do not need to cross long distances to visit a human expert. Instead they could simply chat with a virtual assistant. Another benefit of using chatbots in health care is the scalability of chatbots as they can consult multiple patients at the same time and can consequently reduce personnel costs.

Although it is technologically possible to create chatbots, the community still needs to accept it. Especially in the psychological and medical sectors, users need to trust chatbots to handle critical cases. In 2017 according to *Pega*, the worldwide customer acceptance of the usage of chatbots in health care is 27 percent. Though that rating does not seems to be high, the acceptance of chatbots in health care is valued as the second highest. Only online retail services are higher rated by the respondents [6].

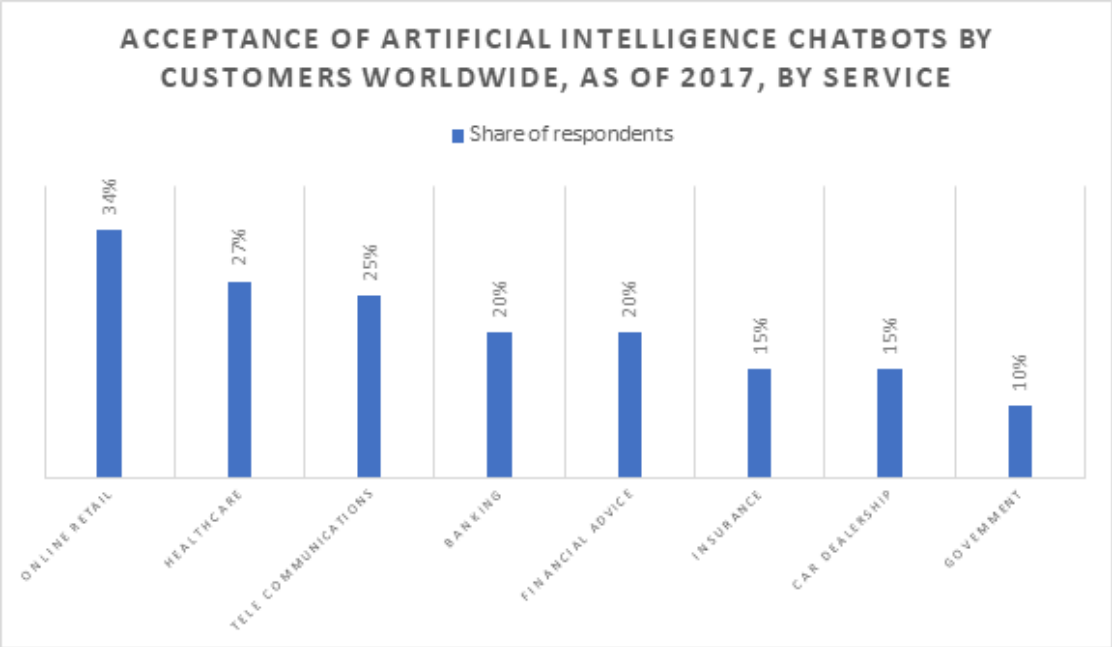


Figure 1.1: Acceptance of chatbots worldwide in 2017 (Own representation based on [6])

The available acceptance of chatbots and the current state of technology build the foundation for the conceptualization and the realization of a chatbot that is especially designed for health care services. Hence, the purpose of this master's thesis is described in the following section.

1.1 Purpose of this Thesis

The continuous availability, instant responses and increased quality of conversations through the cooperation of multiple specialists can be a beneficial impact of the usage of chatbots in health care. However, these assumptions describe the advantages from a patients' perspective. Another advantage from the medical or psychological specialists' point of view can be the reduction of redundant questions that are asked by patients over and over again. Consequently, professionals spend a lot of time on answering redundant questions. Chatbots can reduce the amount of redundant answers directed at the specialist.

The mentioned benefits and the hype about *IBM Watson* create the basic idea of a system that can be used in health care. Examples could be the support of depressive people or making possible diagnosis depending on the users' symptoms. That system has to be adaptable and expandable and should contain the following system components: a mobile application, a back-end and a knowledge base. The mobile application should be responsible for the conversation between a patient and the system and is implemented by the usage of *IBM Watson* services. The back-end and the knowledge base can be used to manage the required data. However, the whole system is not designed with the objective of replacing specialists. Instead it should support professionals and patients.

1.2 Structure of this Thesis

The structure of this thesis is as follows: To improve the readers' understanding about chatbots, the second chapter defines the term *chatbot* and explains its history. Furthermore, it illustrates the reason why services of *IBM Watson* are used during this master's

thesis and the work-flow as well as the structure of *IBM Watson Assistant*. Chapter 3 discusses two applications that represent virtual assistants in the medical or psychological sectors. These apps are used to create functional and non-functional requirements for the overall system that are defined in chapter 4. Depending on the requirements, the concept and the architecture of the system components were designed and partially implemented. This is demonstrated in chapter 5. Chapter 6 describes important implementation aspects of the mobile application to improve the readers' understanding of it. Due to time constraints it was not possible to implement the back-end and the knowledge base nor to use further services that can improve the chatbots' interaction with the user. However, the theoretical approach of possible improvements are discussed in Chapter 7. The results of this master's thesis are concluded in Chapter 8.

2

Fundamentals

This chapter explains the fundamentals, i.e. the definition and the history of chatbots. In addition, IBM Watson and IBM Watson Assistant are described due to their main usage as a chatbot service during this master's thesis.

2.1 Definition of Chatbot

A chatbot is defined as a computer program which is able to process textual or aural natural-language input from a user and to generate meaningful output. This output will be created through the use of *Rule-based Systems* or *Artificial Intelligence (AI)* [7]. The difference is shortly explained in Section. 2.2

To resolve a more accurate result of chatbots they can be combined with search engines or thesauri [7]. As an independent program it can be plugged into multiple messaging platforms, such as *Facebook Messenger*, *Slack* or *Skype* and replace repeated processes [8]. Usually, textual conversation chatbots are combined with *Graphical User Interfaces (GUI)*. They can be represented by animated avatars or can be displayed as a simple chat on a website [7]. However, because of the envolvement of voice technology, companies like *Google*, *Apple* or *Amazon* developed chatbots with aural input which do not need any GUIs. Hidden in a physical device, they are able to understand verbal input from the user and speak the answer in a predefined language. Combined with further functionalities, users can switch on or switch off the lights or play music just by talking to the chatbots [8].

2.2 Rule-Based vs. Artificial Intelligence

Rule-based Systems use rules as the representation of knowledge. That means instead of using static knowledge they use a set of rules that describe what the systems have to do. This kind of systems pertains to the simplest form of AI as it can be created as a set of assertions and a set of rules that define how to act on the assertion set [9]. The following example is shown to improve the readers understanding about the mentioned term: *Rule*.

```
1 <category>
2     <pattern>How are you?</pattern>
3     <template>I'm fine.</template>
4 </category>
```

Listing 2.1: Example of a *category* written in the Artificial Intelligence Markup Language

The *category* that is shown in Listing 2.1 is a rule for matching an input and converting it into an output. The input is represented by the *pattern* and the output is represented by the *template*. The example is written in the *Artificial Intelligence Markup Language (AIML)*. AIML was developed by the *Alicebot free software community* during 1995 and 2000 and is a derivative of *Extensible Markup Language (XML)*. This language should enable people to add knowledge into chatbots based on the *A.L.I.C.E* free software technology [10]. The chatbot *A.L.I.C.E* is described in Section 2.3.

In contrast to rule-based systems, AI chatbots are using methods that refer to artificial intelligence such as neural networks. Neural networks applied to *natural language processing (NPL)* has to transform each word into a numerical vector. Each word that is represented by a vector can be learned by the neural network. The approach of learning algorithms differentiates the concepts of rule-based and AI systems. That means, instead of using predefined rules, neural networks that are trained with millions of parameters can calculate the response to an input by using matrix multiplications and non-linear functions [11].

2.3 The history of Chatbots

The state of the art that is described in Section 2.1 above, is a result of years of research, which is shown in Figure 2.1.

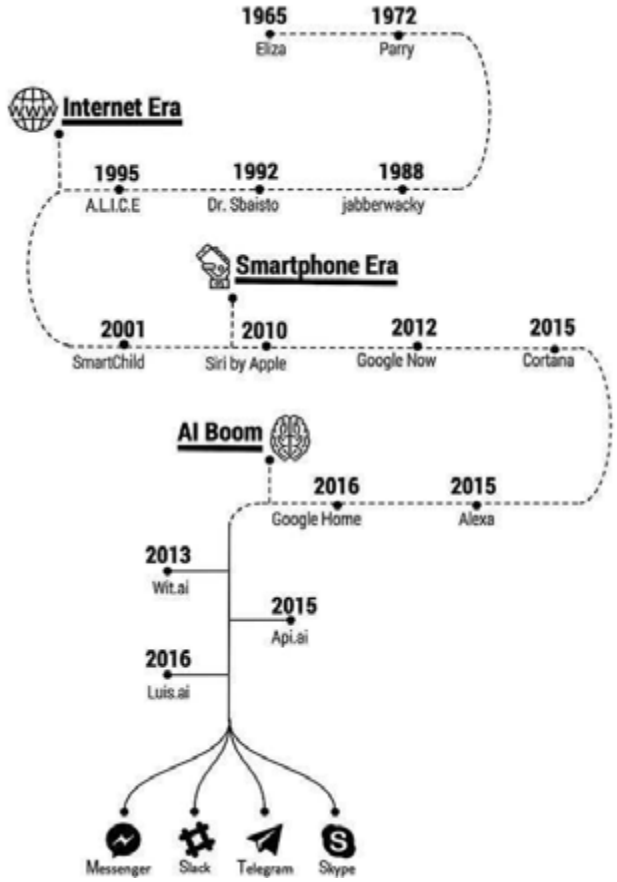


Figure 2.1: History of Chatbots [8]

In 1966, before the first personal computer was established, *Joseph Weizenbaum* created the program *ELIZA* at the *Massachusetts Institute of Technology (MIT)*. He developed the first-ever chatbot, which was build to simulate a psychatrist [8]. According to *Joseph Weizenbaum* the program analyzed input sentences based on decomposition rules which were triggered by key words appearing in the input text. The answers were generated by rules, which were related to the decomposition rules [12].

After the development of *ELIZA*, *Kenneth Colby* developed a new chatbot, called *Parry*. The psychatrist created a chatbot, which simulated a patient who suffered from paranoid schizophrenia [8]. *Parrys*' architecture was similar to its precursor *ELIZA*. However, it contained a state of mind as well as the knowledge about the conversation. These two aspects allow the program to generate responses which are not only influenced by the input, but also by Parrys' desires and beliefs. The developer collected records of the conversations between psychiatrists, patients and *Parry* and presented them to another group of psychiatrists. Afterwards he asked his colleagues if they could find out if the answers came from the program or the patient but they could not [13].

About 20 years later in 1995 another well known chatbot was developed by *Richard Wallace*. It was called *Artificial Linguistic Internet Computer Entity* or simply *A.L.I.C.E*. It was still based on *pattern matching*, like *ELIZA*, but differs in that it tries to reflect more human behaviour. The basic idea of this program was to talk as long as possible, so the users would not realize that they were talking to a machine [13]. *A.L.I.C.E* was awarded the *Loebner Prize* as one of the strongest chatbots in history [8].

The *Loebner Prize* was founded by *Hugh Loebner* and the *Cambridge Centre for Behavioural studies* in 1991 and is one of the oldest *Turing Test* contest in the world [14]. In 1950, the British mathematician *Alan Turing* investigated the question whether a machine is able to think. This question was the foundation of the *Turing Test*. To evaluate this issue, he designed a method to investigate if a machine can demonstrate human intelligence. In order to test his hyptohesis, one participant had to communicate with two other participants via a computer terminal. One participant was a human, the other one was a machine. After the conversation ended, the participant had to identify who was a human being and who was a machine, based on the quality of the dialog. In addition, *Hugh Loebner* set the rule in place, that four judges had to communicate with a program for 25 minutes. As soon as more than 30% of the jury members considered the machine as being human, the test was passed [15]. After 27 years, the results of the competition in 2018 shows that machines are not still able to act completely like a human being. The machines received ratings between 23% and 33%. The rating demonstrated how human-like the machines acted [14].

In 2001, the company *ActiveBuddy* developed the chatbot *SmarterChild*. Because of the usage of *MSN Messenger* and *AOL Instant Messenger*, whose user numbers had at the time risen to 30 million, the program was extremly common. The difference of the chatbot, compared to its precursor, was the provided information. It was not only built to amuse the user, but rather to offer information about sports scores, movie quotes etc. [8].

SmarterChild is the precursor of *Siri*. The *Apple* product Siri is a personal assistant which is able to understand aural input and offers useful information or supports the user in daily life [8]. For example, it allows the user to get the current weather forecast, play music, or to translate text [16].

The continuous advances of scientific knowledge reduces technological limitations and increases the interest in chatbots. Besides *Apple*, other corporations such as *Google*, *Amazon*, *Microsoft* or *Facebook* are using chatbots as well [17]. Based on the grown interest in the subject, the usage of chatbots has increased extremely in the last few years. This can be shown by the following figure.

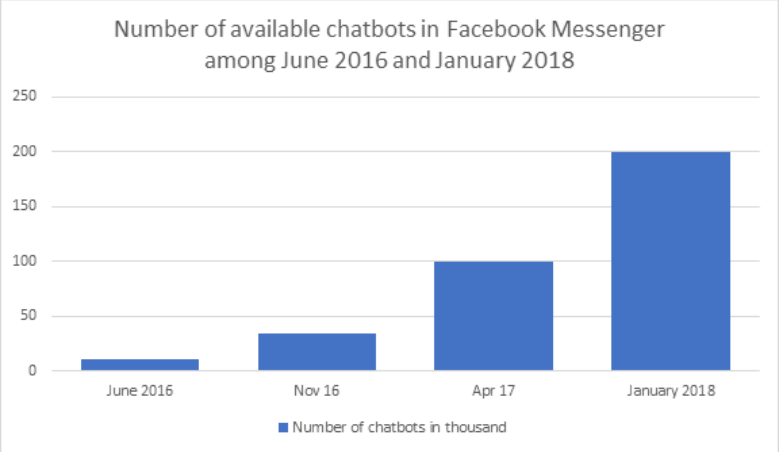


Figure 2.2: Number of chatbots in *Facebook Messenger* between June 2016 and January 2018 (Own representation based on [18])

According to *Khari Johnson*, the messaging application *Facebook Messenger* contained about 200000 chatbots in January 2018 [18]. As a dominating player in messaging services, *Facebook* invested enormously in communication bots. One important project

for *Facebook* is *M*. The virtual assistant is not only an automatic tool but is also backed up by humans. Hence, whenever it does not know the answer to a user input, it receives assistance by a human [17].

Besides the mentioned companies, *IBM* is also a big player in reference to chatbots. The corporation and the hype about *IBM Watson* will be described in the next section, as it is used as the main conversation service during this master's thesis.

2.4 IBM Watson

In February 2011, the big hype about *IBM Watson* started with its win in the television game show *Jeopardy!* against the two biggest all-time champions. The game includes three rounds with different categories. Each category has five answer, which are sorted by difficulty and their amount of money, starting from 100$ until 500$. Each of the three contestants, including the returning champion, needed to answer a choosen statement in question form. The returning champion will start with the statement. After he had choosen a statement, the fastest participant in pressing the buzzer afterwards was allowed to present a question. If the given question was correct, the amount of the answer will be added to his account. However, if the given question was wrong, the amount will be substracted from his account [19]. As it is shown in Figure 2.3, the team of *IBM Research* accepted the challenge, to compete against *Ken Jennings* and *Brad Rutter*, "the two most successful players in the quiz show's history" [20], with *IBM Watson*.

IBM Watson is defined as a computer running software called *DeepQA* [21]. This stands for "a software architecture for deep content analysis and evidence-based reasoning that embodies that philosophy" [22]. The system has a power of 2880 processor cores and runs on a cluster of 90 servers. The reason to build such a system was, besides winning the competition, to create a new generation of technology which is capable of finding answers in a huge amount of unstructured data. According to the principal investigator for the *DeepQA* project *David Ferrucci*, the software *Watson* is however neither desgined to model a human brain nor work like one. Rather, it should be a new technology which

is more effective in understanding and interacting in natural language than common technologies [21].

Figure 2.3: *IBM Watson* against *Ken Jennings* and *Brad Rutter* [23]

To prepare the system for the game show, the team fed it with millions of information. It contained documents, encyclopedias, dictionaries, novels, taxonomies and religious texts [24]. The whole learning process took years [21]. As a result of its ability to analyze the input and give the correct answer faster than its rivals, IBM Watson was able to compete in the game show successfully.

The whole process that makes *Watson* able to understand the input, spoken in a natural language, and giving a meaningful answer is just simplified above. Actually, more than one hundred algorithms analyze the input at the same time and in different ways. All of these algorithms find possible and plausible answers for the question. For each answer the system finds different evidence which can refute or support the potential answer. Another set of hundreds of algorithms score the degree to which the found evidence approves the answer. The system replies with the answer which has the highest ranking. For *Jeopardy!*, *IBM Watson* was trained to buzz only if the highest rated rank is high enough. This strategy made it possible to avoid that *Watson* loses money by telling the wrong answer [21].

As well as winning the game show and showing people how powerful AI based systems can be, *IBM* has further goals. *IBM Watson* can be applied to different areas such as healthcare, shopping or travelling. According to *IBM*, *Watson* could be established as an online tool which is able to diagnose patients and help medical professionals. For example, the system can help a doctor to establish a final diagnose by providing him with different diagnoses depening on a set of symptoms or the medical history [24].

This statement was essential for using *IBM Watson Assistant* as a conversation service during this thesis, which is described in the following section.

2.5 IBM Watson Assistant

IBM Watson Assistant is a single service of a big collection of services, provided by *IBM*. It allows the user to create a chatbot for specific purposes in a simple way. According to *IBM*, an assistant is a cognitive and customizable bot for business needs [25]. The use of other services can help to make a chatbot more intelligent. These services and their possible usages are discussed in Chapter 7.

*IBM*s conversation service can be integrated into custom applications, as it is shown in Figure 2.4.

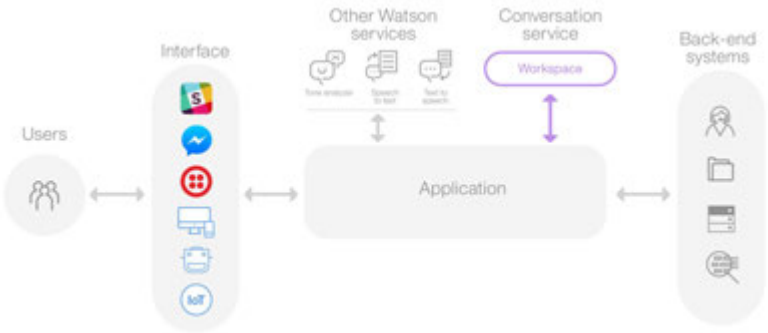


Figure 2.4: System architecture of a custom application using *IBM Watsons Assistant* [26]

The figure above shows that users can interact with a custom application via a predefined or new designed *GUI*. The application itself contains the logic. It is connected with a workspace of the conversation service or other *Watson* services [26].

Since the 9th of November 2018, the term *workspace* has changed to *Skill*. A skill can be defined as a container, which holds the training data and the structure of the dialog and allows the assistant to understand the natural language input and responses to it in a useful way [27].

To be more precise, each skill contains three different types of components, also called artifacts, which are described in the following sections.

2.5.1 Intent

An *Intent* can be compared with the purpose of the users' input [28]. For example, the user wants to know the weather forecast, which would be the *Intent*. In order to do so, he might ask "What's it like outside?" or "How's the weather?". Different questions but asked for the same reason, are defined as *Example*s. They cluster the possible user inputs, which refer to the same topic. To make a chatbot more precise, it is relevant to define *Intent*s as accurately as possible. An *Intent* which is called "WEATHER" and which contains *Example*s such as "How's the weather going to be tomorrow?" and "Is it a beautiful day for a walk?" make it impossible to respond in a meaningful way. Although both questions refer to the same topic, they must be handled separately. To distinguish between these questions, the developer can add multiple *Intent*s to one single *Skill*.

2.5.2 Entity

Entities can be very helpful to improve the chatbot further. An *Entity* represents a term which is relevant to an *Intent* [28]. A chatbot which should suggest local restaurants to the user could be created by feeding the knowledge base with a lot of examples. However, the data would be extremely scaled up because each possible case must be covered with an example. This is demonstrated in the following:

Possible *Example*s for Italian food:

- "I'd like to eat some **pizza**."
- "Tell me, where can I order **Italian food**."
- "I'd really love it, if I could eat **pasta** tonight!"

Possible *Example*s for Asian food:

- "I'd like to eat some **Sushi**."
- "Tell me, where can i order **fried chicken with rice**."
- "I'd really love it, if I could eat **Chow Mein** tonight!"

As demonstrated in the sample above, the *Example*s remain the same, but the context changes. Determining the right context is necessary for a meaningful response. The usage of *Entities* can be a solution for this issue as they are combined with *Intent*s.

An *Entity* contains one or multiple *Entity Values*, which represent synonyms or patterns. Using the following *Entities* with synonyms instead of creating a huge set of examples could solve the issue above:

Entity	Entity Value	Synonyms
@Asia	Dishes	"Sushi", "Chow Mein", "Khao Man Gai Thai Chicken"
@Asia	Chicken	"fried chicken", "pullets", "crispy chicken"
@Italian	Dishes	"Pizza", "Pasta"
@Italian	GeneralTerms	"italian food", "Italian"

Table 2.1: Possible *Entities* to reduce the amount of *Examples*

Patterns can be used to detect e.g. e-mail addresses. After defining *Intent*s and *Entities*, they can be applied to a dialog, which is described in the following section.

2.5.3 Dialog

IBM Watson Assistant offers the opportunity to create *Intent*s, *Entities* and a completely customized dialog via a graphical user interface.

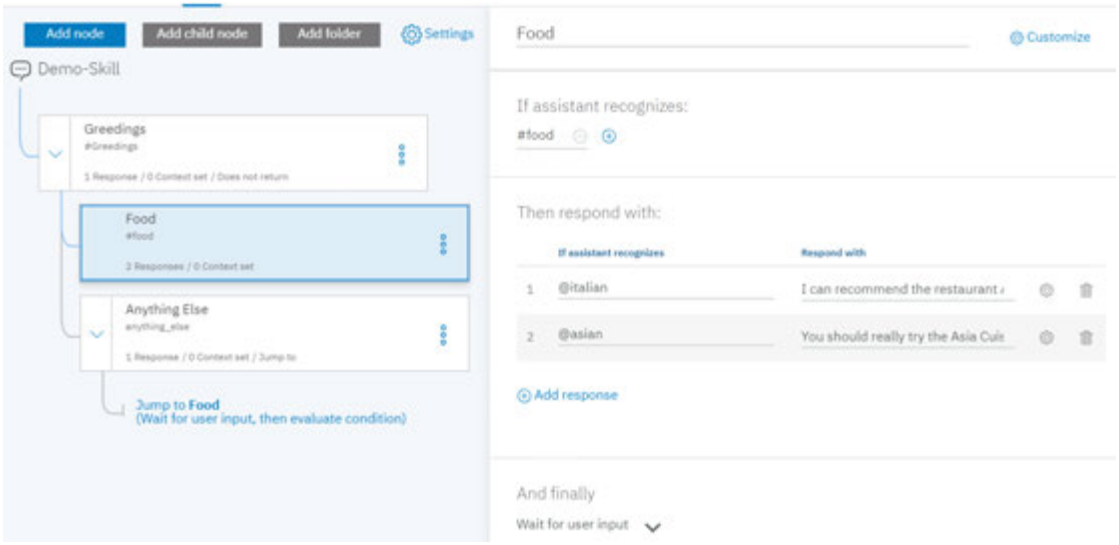
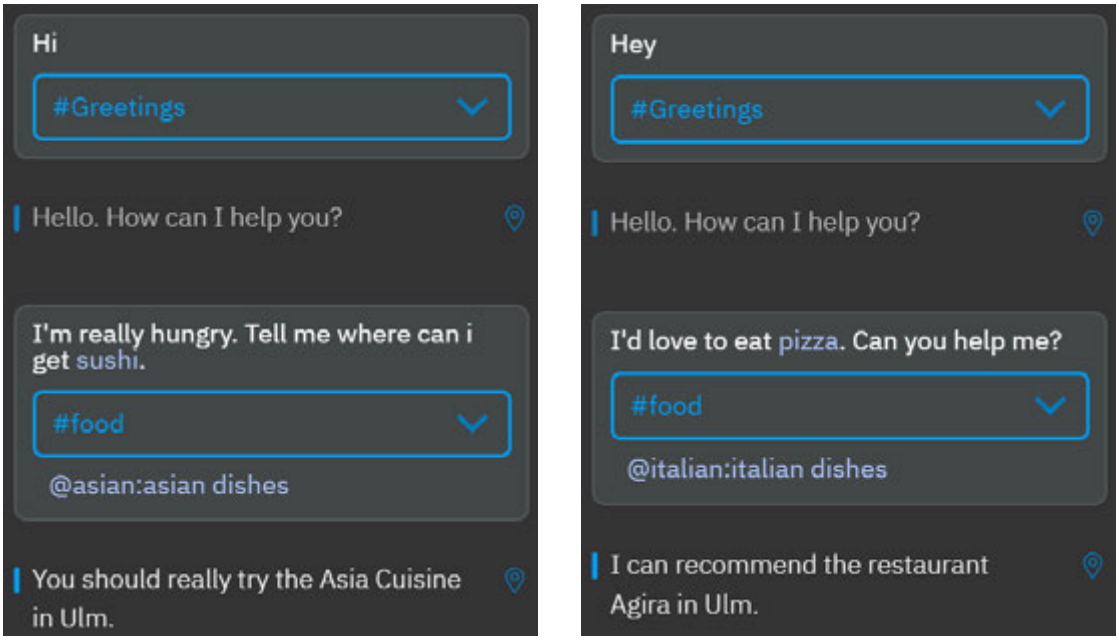


Figure 2.5: Demo-Dialog created with *IBM Watson Assistant*

A dialog can be designed as a tree structure, as it is shown in Figure 2.5. Each dialog node has a set of properties. The first property is the condition, that tells the system how it should be evaluated. In the example above, the dialog node with the title "Food" will be activated if the system recognizes an input that matches with one of the examples of the *Intent* "#food". The prefix "#" indicates the word "food" as an *Intent*. Furthermore, *IBM Watson Assistant* allows the developer to handle different recognition in one node by defining multiple answers for multiple *Entities*. An example is shown in Figure 2.5.

Starting from the beginning, the first dialog node will give the predefined answer "Hello. How can I help you?" whenever the user puts in a phrase which matches with an *Example* of the *Intent* "Greetings". Now, asking for a restaurant the next *child node* is trying to recognize any match with the given input. However, this can only happen if the dialog has a *context*. A *context* maintains state information about a conversation. This kind of data can be passed between an application and the *Watson Assistant* service. Without maintaining the state information, the conversation would start from the beginning each

time [29]. The conversation in Figure 2.6 has a *context*, that means it can go forward after the greetings. The sample shows, that the *Intent* "food" is recognized for both conversations. However, graphic **(A)** in figure 2.6 shows how the chatbot reacts after asking for Sushi. The word "Sushi" is highlighted because it is recognized as a synonym for the *Entity Value* "Asian dishes". In addition, in graphic **(B)** the *Entity Value* "Italian dishes" is recognized. According to that, the predefined response changed as well.



(A) System recgonizes *@Asian* (B) System recgonizes *@Italian*

Figure 2.6: Comparison of different dialog instances

The sample above can also demonstrate that *IBM Watson Assistant* is able to allow deviations between the defined *Example*s and the user input. The *Intent* "food" contains the *Example*s "I'd like to eat" and "I'm hungry. I'd like to have" which departs from the original input in Figure 2.6. However, if the user input can not be matched with any condition in the current dialog node, the next node is going to be evaluated. In this case, the next node represents an *Anything Else* case. This case is available by default, so the developer does not need to create it first. The *Intent* "anything_else" is triggered if no other condition is recognized. Figure 2.5 shows a dialog node with the title "Anything Else" which represents a fallback. If the user input is completely unknown, this node is evaluated. After a reply, which can be predefined as well, the node will reference to the

dialog node above and waits for a new user input. This behavior is possible, because *IBM Watson Assistant* allows the developer to control, how the dialog node should act after its execution. After a *DialogNode* has been evaluated, three different predefined options are presented.

- **Wait for the user input**
- **Skip user input**
- **Jump to**

Besides a gradual procedure, this allows the developer to create loops for keeping up the conversation or skip parts of the dialog if they are not necessary. The described components of *IBM Watson Assistant* allow the developer to create a chatbot service without coding. As written above, it is used as the conversation service in this master's thesis.

As already mentioned, other companies besides *IBM* provides solutions for the conversation between machines and humans as well. The skills of the services depend on the content and structure of its knowledge base. That means that chatbots could assist users in the travel industry as well as in the food industry. As this master's thesis refers to the psychological and medical sectors, the following chapter demonstrates the usage of medical or psychological assistants.

3

Related Work

Based on long term research and the continuous advancement of chatbots, the amount of different use-cases of chatbots is growing constantly. This refers to health care as well. For example, the developed chatbot *Smart Wireless Interactive Healthcare System (SWTCHes)* can be used to counteract obesity and overweight as these diseases present the fifth leading risk for global deaths. A user can enter his goal weight as well as his current weight and the system creates a feasible plan. Furthermore, during a conversation with the chatbot the user can receive information about diets and exercise plans [30]. Besides the mentioned use-case, chatbots can also be used to help people with occupational stress. Therefore, conversational services can be used to learn an indivudial's stressor profile, to offer personalized peer support and to help people with their individual stress [31]. Another approach of using chatbots in health care can be to help people quit smoking. Members of the *City University of Hong Kong* showed that the presence of conversational agents increased the participant engagement and and enhanced them to quit smoking. Therefore the chatbot sent automatic reminders or engaged in conversations. Users were able to share experienced flavorable effects among each other and the chatbot also sent valuable health information related to smoking to the users [32]. Besides diseases and bad habits, chatbots could also be a solution for loneliness. As chatting with people is a basic requirement especially for elderly people, a conversational service was developed in Taiwan that simulates a conversation partner [33]. The following Figure 3.1 shows a dialogue between elder women an the conversational agent that is represented as a human head.

In contrast to the use-cases above, the chatbot that is developed during this master's thesis is represented through a mobile application. Therefore, the two mobile applications

Figure 3.1: Conversation between elder woman and a conversation agent [33]

Woebot and *Babylon* that represent conversation agents in the psychological or medical sector are tested more accurately.

During discussions about this master's thesis, *Woebot* was mentioned multiple times. Hence, *Woebot* was investigated to get an impression of how chatbots relating to the psychological sector work. As this thesis obtains to the medical sector as well, another mobile application was needed to get an impression how chatbots can handle medical issues. A web search led to the mobile application *Babylon Health* as it belongs to "The Top 12 Health Chatbots" according to *The Medical Futurist* [34].

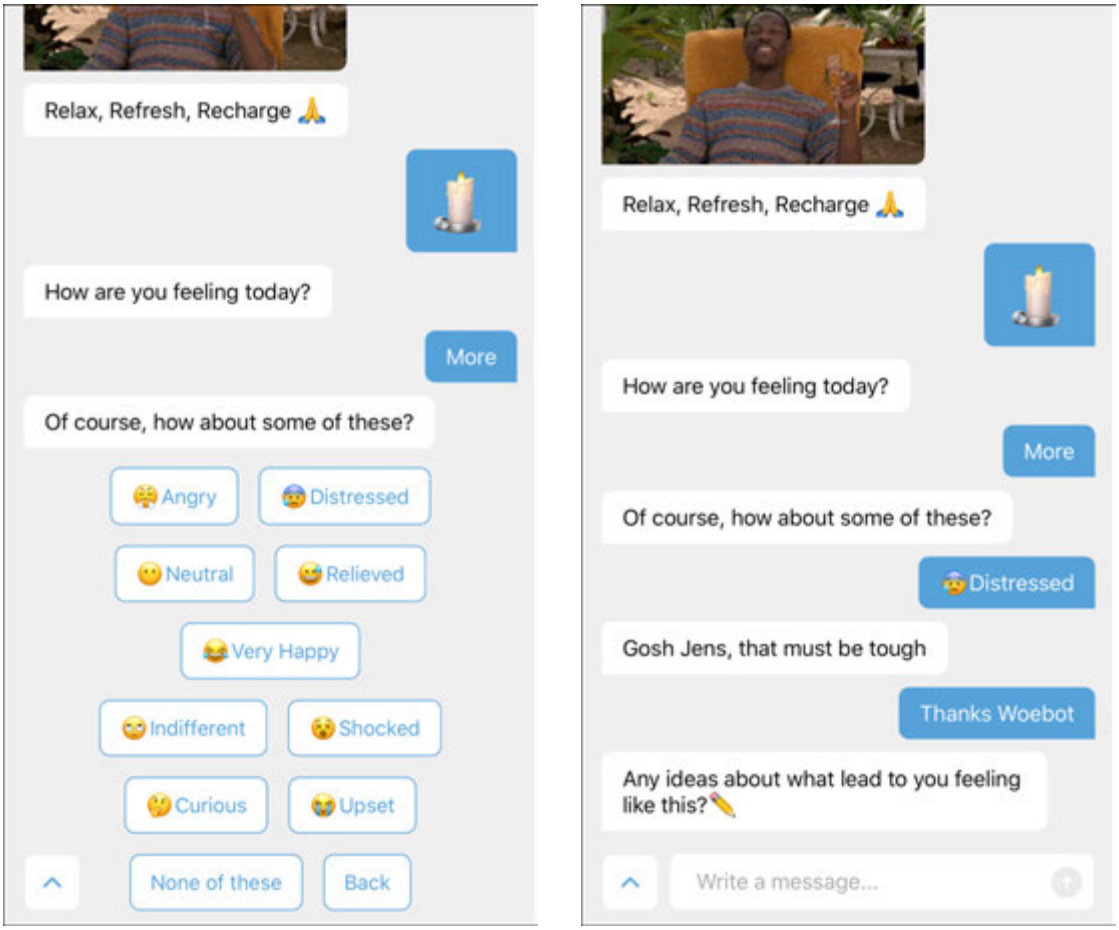
In general, the mentioned chatbots are designed to improve the mental health of the user or to provide trial and health information to him. Both applications are illustrated in the following sections. The last section discusses the similarities as well as the different features of each application. The result of the investigation is used for the requirements of the overall system.

3.1 Woebot

Woebot! is a chatbot that is supposed to help people with their mental health by creating a friendly and informative conversation [35]. Therefore it asks users about their feelings and what is currently going on in their daily lives. The chatbot also sends videos or other tools to the user depending on the patients' mental state [36].

The application is the result of a combination of psychological expertise, sense of humor and natural language processing. To test the benefit of the chatbot, the procuders conducted a study at Stanford University [36]. Therefore 70 individuals aged between 18 and 28 years were recruited. Each participant received a self-help content derived from Cognitive behavioral therapy (CBT) principles. The CBT principles were designed either in a conversational format, that means as the conversational agent *Woebot*, or as the ebook "Depression in College Students". While the control group occupied themselves with the ebook, that contained frequently asked questions about depression for two weeks, the participants of the experimental group communicated with *Woebot*. The objective was to assess the acceptability, the efficiency and the feasibility of a conversational agent that delivers a self-help program for college students who have symptoms of anxiety and depression [37]. The result of this study was that the use of *Woebot* led to a significant reduction of anxiety and depression compared to the information-only control group [35].

The mobile application *Woebot* was used for several days during this master's thesis to get a personal impression about it. As a result of that, certain characteristics were detected and are discussed more accurately in Section 3.3. However, to demonstrate *Woebot* and improve the readers' understanding of it, a small part of a conversation with the chatbot is displayed in the following Figure 3.2. The whole exchange can be described as a mix between a guided and a free conversation. Depending on the entered question, the mobile application provides predefined answers. This is demonstrated in graphic (A) of Figure 3.2. Furthermore, it gives the user the opportunity to enter customized answers.



(A) Predefined answers (B) Custom answer

Figure 3.2: Conversation with *Woebot* with different types of answers

To describe the scenario in image (B) a bit further, the chatbot asked the user what he is currently doing. The answer "Just relaxing" led to the video that is displayed at the top of graphic (B). So this demonstrates one of multiple videos that can be sent to the users to brighten them up.

Besides *Woebot*, further applications that represent a virtual doctor or a psychotherapist exist. Another example is demonstrated in the following section.

3.2 Babylon

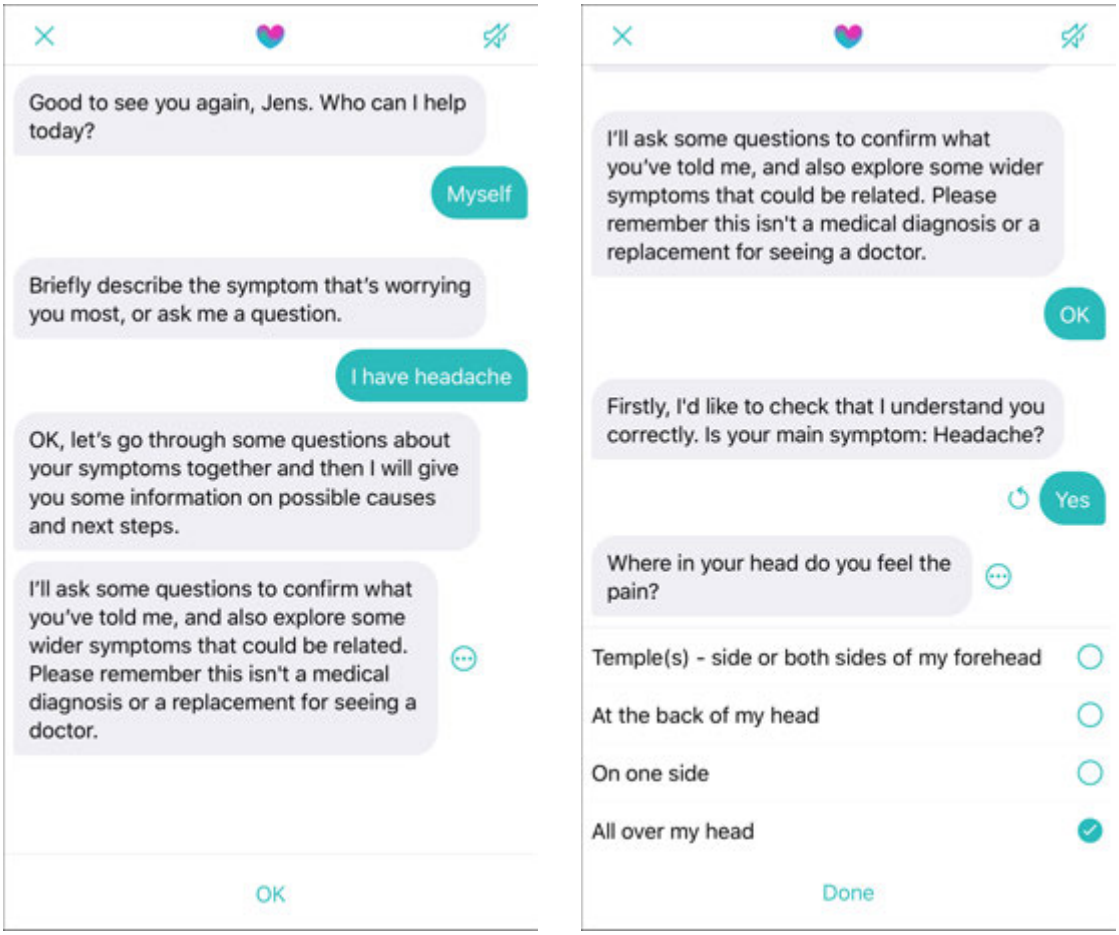
Babylon Health is a mobile application that provides health and triage information depending on the patients symptoms [38]. A patient needs to enter a symptom and must pass a guided conversation. During this conversation, the system tries to get more precise information about the users' state of health. After the procedure is finished, the application lists possible causes to the entered symptoms.

To get a personal impression as well as the similarities and the differences to *Woebot*, the application was tested for several days during this master's thesis. The test was conducted as follows: Before the application can provide health information, the patient needs to enter a symptom. To test the system, the symptoms "headache", "cough" and "feeling mournful" were chosen.

The following Figure 3.3 displays the beginning of a conversation by entering the phrase "I have a headache". The answers "Myself", "OK", "Yes" are predefined as it is shown in Figure 3.3. An answer can be chosen by selecting and confirming it. A progress bar at the top of the view displays the amount of the open questions that can still be asked by the system. Furthermore, the mobile application provides a functionality to take back and adapt an answer as well as to rate a conversation. This feature is discussed more precisely in the following section as it is necessary for the system that was created during this master's thesis.

After a conversation starts, the system explains the user that it does not provide a medical diagnose neither it replaces the visit of a doctor. However, it allows a patient to make a medical appointment. Therefore, the user can choose the date and time of the appointment and can decide whether it should be a video or a phone call. To complete the booking process, the patient needs to describe his symptoms.

Another feature that is provided by the app is the *Healtcheck*. The user can get a health report or practical insights to stay healthy by answering questions about his or her lifestyle and family history [39].

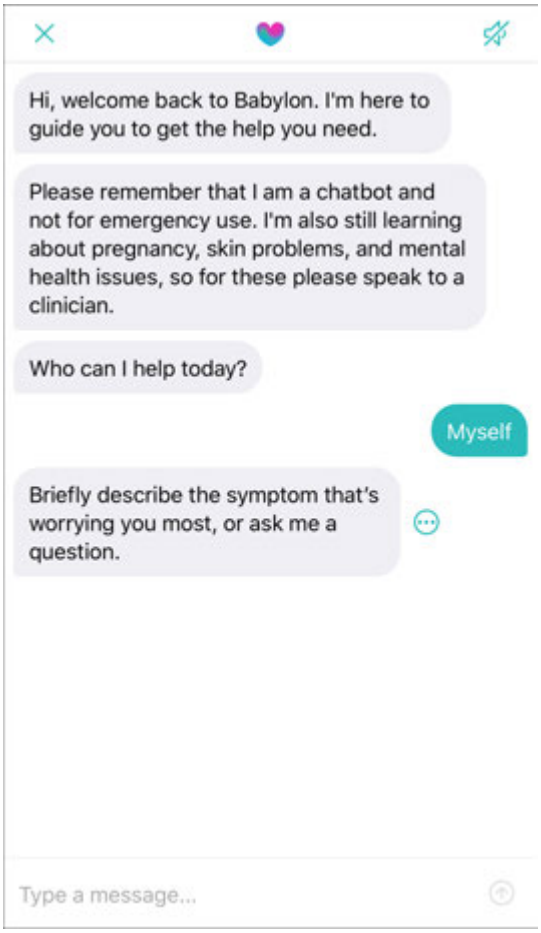


(A) Start of a conversation about headache using *Babylon Health*

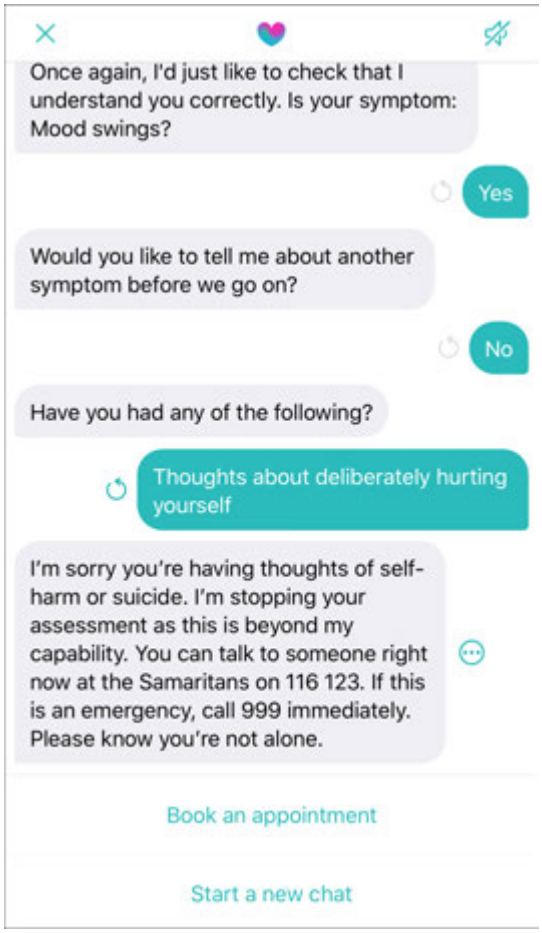
(B) Isolating the symptoms with *Babylon Health*

Figure 3.3: Conversation with *Babylon Health* about the symptom *headache*

According to the *Babylon Health* itself, the app is not made for emergency use and mental health issues. However, to test how it handles the detection of suicidal thoughts, a conversation was started with "I feel pretty sad". The conversation ended up with the last speech bubble that is shown in Figure 3.4 graphic (B). The system advises the user to call 999 or look for help at *Samaritans* and explains that this situation is beyond its capability.



(A) *Babylon Health* notes the user about its limitation



(B) The end of a conversation containing depressive symptoms

Figure 3.4: Conversation with *Babylon Health* about depression

Although this application does not handle medical emergencies directly, it demonstrates that it is necessary to catch exceptions and provide help to the patient. This aspect is discussed besides other relevant features in the following section.

3.3 Discussion

Both illustrated systems provide a conversation to help users with their mental health or to inform patients about their physical condition. As one relates to mental health and the other one to physical health, differences between both applications exist. Whereas

Woebot tries to stay in a continuous dialog with the user, *Babylon Health* represents a symptom checker that can be used to receive health information depending on the users' symptoms. However, both conversations are formatted as a, at least partially guided conversation. That means that answers are not completely but mostly predefined. Besides the similar structures of the dialogue, emergency cases are handled similarly, too. Both apps provide information to the patient about institutes that can help the user in the recognized emergency. As a result of that, the user still needs to become active by e.g. calling a specialist. Significant distinctions can be recognized as well and refer to the usability of the apps. As already mentioned, *Babylon Health* provides multiple features such as rating a conversation or a single system response as well as to revoke an answer. This functionality could not be found in *Woebot*.

The mentioned aspects are used to define requirements of the overall system. As the system should cover the medical and the psychological sector, the concept has to be generic. That means that the system should be able to handle different use-cases. However, it should be kept in mind that the system can only be useful to the user if it contains a large amount of knowledge. Without that, the system would not be able to reply meaningfully.

4

Requirements

The following chapter describes the system requirements of the mobile application and the back-end application. In addition, the requirements of all involved user-roles are defined as well.

The first section describes the functional requirements of the mobile application. Afterwards, the non-functional requirements of the mobile application are defined. Although the back-end application is not implemented during this master's thesis, the requirements are defined as they are necessary to create a theoretical concept of the knowledge base and the back-end application. This is described in Section 4.3. Furthermore, human *experts* are involved in the system and need to act in critical cases. Therefore, it is necessary to define requirements for their action as well. This is described in the last Section 4.5.

4.1 Functional Requirements of the Mobile Application

The user roles *Expert* and *User* have to be handled separately as their usage of the mobile application is different. Therefore, the requirements need to be split as well. The first section describes the general requirements which are the same for both roles. The sections 4.1.2 and 4.1.3 describe the separated functional requirements.

4.1.1 General Functional Requirements of the Mobile Application

FR_01 - Registration: The application must provide the functionality to allow the users or experts to create an account. Therefore the following credentials are required:

- Gender
- E-Mail
- First name
- Last name
- Password

FR_02 - log-in: The application must provide a functionality to allow the users or experts to log-in. The user should be only able to log-in after creating an account.

FR_03 - log-out: The application must provide a functionality to allow the users or experts to log-out. After a log-out, the application must provide a functionality to let the user sign in again.

FR_04 - Differ different roles: The application needs to differentiate between the two roles *USER* and *EXPERT*. This includes different views and system interactions. The named roles are explained more detailed in chapter 5.

FR_05 - Multilingualism: The application must support multilingualism. This requirement includes the language of the mobile application and the output of the chatbot.

FR_06 - Adapt wrong system output: The user input can be classified incorrectly by the underlying chatbot system like *IBM Watson Assistant*. To handle this issue, the user needs to be able to mark wrong output as unclassified.

FR_07 - Rating the system output: The contextual precision and the error rate depends on the content, structure and the scope of the knowledge base of a chatbot service. To counteract a big error rate, the mobile application needs to provide a functionality to rate any system output. If a system output has an average assessment below e.g. 70% it is sent to an expert, who can adjust the predefined answer. This requirement could not be implemented completely because of time conditions.

4.1.2 Functional Requirements of the Mobile Application for the User

FR_08 - Textual interaction with the system: The system must provide a view to allow communication between the user and the system. It must provide the functionality that

allows the user to write and send a question to the system. The system needs to display the answers to the user as well.

FR_09 - Send Push-Notifications: The application must be able to send push notifications to the experts. Notifications can be sent whenever a special case occurs. A special case can be an escalation which represents a critical case such as the detection of suicidal thoughts, a medical emergency or if the system needs human assistance because it does not understand the users' input.

FR_10 - Multiple topics: The application must be able to provide multiple topics to the user. After selecting one, the user can ask specific questions about the topic to the system.

FR_11 - Handle unknown user input: The system must be able to detect and handle an unknown user input. An input is defined as unknown if the chatbots algorithm can not find a predefined answer to the question.

FR_12 - Detect escalations: The system must be able to detect escalation cases. An escalation case can occur if e.g. the user talks about suicide or depression.

4.1.3 Functional Requirements of the Mobile Application for the Expert

FR_13 - Registration: Experts are responsible for the content of the knowledge base. Consequently, unlike the users' registration process this registration is only completely finished if the provider approved the experts application. The whole procedure is described more precisely in Section 5.2.

FR_14 - Receive Push-Notifications: The application needs to receive notifications, sent by the users phone after a special case has been detected. A special case can be the detection of suicidal thoughts or if the user input is unknown to the system. After the expert received and opened a notification, the expert can see the chat history between the user and the system.

FR_15 - Handle escalations: The escalation triggers a notification which is sent to the expert. Afterwards, a new communication channel between the users' mobile application

and the experts' mobile application should be opened. Then the expert can communicate with the user directly.

FR_16 - Extend knowledge base: Whenever the user receives and opens a notification, referring to an *unknown input case (FR_11)*, the application asks the expert about the necessary data to extend the knowledge base. After the experts run through this process successfully, the chatbot is able to answer the previously unknown user input.

4.2 Non-Functional Requirements of the Mobile Application

NFR_01 - Stability: Being an efficient application the mobile application needs to be as stable as possible. Especially in relation to escalation cases, push notification must be sent successfully to an expert. If it is not possible, e.g. if the device does not have an internet connection, the notification must be sent as soon as the issue has been fixed. Crashes during a chat must be intercepted, too. Due to time conditions, this non-functional requirement could not be implemented completely.

NFR_02 - Usability: It should be easy for the user or the expert to handle the mobile application. The system responses should be predefined in a user-friendly way to simplify the conversation between a human and the chatbot.

NFR_03 - Generic: Services like the *IBM Watson Assistant* are still evolving. The application should be able to adapt to this enhancements. Therefore it is necessary to keep the system concept and architecture generic.

4.3 Functional Requirements of the Back-end Application

The back-end application is only necessary for the *expert*. Therefore, the requirements do not need to be separated for different user roles.

FR_17 - Provider-Notification: The expert can only adjust or extend the knowledge of the system if he is successfully proven by a *provider*. Consequently, *providers* need to

be informed by the system after an *expert* started a registration process. This procedure is explained more precisely in Section 5.2.

FR_18 - log-in: The application must provide a functionality to allow experts to sign in via a graphical user interface. The expert should only be able to log-in after becoming a part of the expert-team.

FR_19 - log-out: The application must provide a functionality to allow the experts to log-out. After a log-out, the application must provide a functionality so the user can sign in again.

FR_20 - Get data: The application needs to provide a functionality to request data from the back-end. This data relates to the knowledge of the chatbot, user credentials and chat histories. Furthermore it is necessary to handle the situations that are described in *FR_11* and *FR_12*.

FR_21 - Update data: The application needs to provide a functionality to update data on the server. The definition of data is equal to the definition in *FR_20*.

FR_22 - Delete data: The application needs to provide a functionality to delete data from the server. The definition of data is equal to the definition in *FR_20*.

FR_23 - Access Rights: The application must be able to assign access rights per user and per topic.

4.4 Non-Functional Requirements of the Back-end Application

NFR_05 - Graphical user interface: The back-end application should provide a graphical user interface to allow the expert to get, update or delete data of the knowledge base in a simple way.

NFR_06 - Usability: The graphical user interface that is described in *NFR_05* has to be user-friendly so the expert can easily add, update and delete data to the knowledge bases.

NFR_07 - Modularity: The back-end application needs to be implemented through the use of smaller single modules. This approach leads to more flexibility and less redundancy.

NFR_08 - Stability: The back-end application needs to guarantee that data can not be lost even if system errors occurs.

4.5 User Requirements

UR_01 - Expert qualifications: The expert needs to establish specific qualifications. An expert can only be responsible for a specific topic if he has special competences. These qualifications need to be proved by a *provider*. The single user roles are explained more precisely in the following Chapter 5.

UR_01 - Expert behavior during escalation: The expert needs to react in a predefined time limit after he received a Push-Notification. This time limit must be definable per notification as different situations have different priorities.

All described requirements build the foundation of the concepts and system architectures, which are described in the following chapter.

5

Concept and Architecture

In critical situations like the detection of cancer, patients can ask questions repeatedly to a specialist to reassure themselves. However, the time of the experts is usually limited and valuable. Answering redundant questions leads to less time for important issues. The usage of chatbots could be a possible solution for this matter.

This chapter introduces a system with the goal to assist professionals as well as patients in medical or psychological sectors. As a conclusion, instead of visiting a doctor or a psychologist a patient can talk to a virtual assistant.

This chapter exemplifies the concepts and architectures of such a system. The first section describes an overview of the complete system and the interaction between the participants and system components, which is shown in Figure 5.1. The single components of the mobile application and the server application are explained more accurately in Section 5.2 and 5.3.

5.1 Overall System

As mentioned in Chapter 2, the usage of chatbots can be useful in completely different sectors. The following concept refers to the medical or psychological area and must deal with critical aspects like medical emergencies.

To handle such cases, the system needs different types of participants. Besides common *users*, it is necessary that *experts* and *providers* are included. A *user* is defined as a human being, who is using the application without any expert knowledge. As an example, the *user* Steve can be a person, who is affected by depression. He feels weak and

sluggish but he does not know that he is sick. Before he goes to a psychologist, he wants to ask the digital assistant and starts the mobile application. After the mobile application appears on his smartphone, it asks him to create an account. Afterwards, the app shows him multiple topics. Choosing a topic forwards Steve to a chatbot which has knowledge about a specific subject. He selects "Depression" and is immediately greeted by the chatbot. The chatbot asks how it can help Steve and he writes down his symptoms. The system responses with a possible diagnose and keeps talking for a while.

A few days later, Steve communicates again with the chatbot because he recently became really sad. He tells the chatbot that he thinks about his own suicide. The system recognizes this input as an *escalation case* and sends a notification to an *expert*. In this context, an *expert* is defined as a human being who has a high level expertise regarding to a specific topic. As soon as he receives the notification, the system creates a separate communication channel between the *expert* and Steve. Now, the *expert* can talk directly to Steve and can try to help him with his problems. The advice was helpful and after the conversation with a human expert, Steve knows how to handle his issues in the future.

The described scenario relates to one possible case where patients can be supported by the system. Besides the example above, another possible use-case of this system in health care can be the support of overweight people in daily life. Therefore, the chatbot can provide information about workout techniques, recipes of healthy food to the user or encourage him by sending daily messages. The experts can be represented by nutritional advisers or fitness trainers who can extend the knowledge base constantly. Another approach can be the usage of the system as a symptom checker of physical complaints. Patients can tell their self identified symptoms and receive possible causes. Experts can be represented through medical specialists who need to extend the knowledge base of the system or to talk directly to the patients. During a direct conversation, the previous received information during a symptom check can be told to the medical specialist. Furthermore, experts need to be available to react immediately to medical emergencies like a possible diagnose of cancer. However, regarding all possible use-cases real emergencies can not be dealt with this system itself. As soon as critical cases are identified by the system, human assistance is always needed.

The mentioned use-cases present a subset of all possible use-cases of this system. As the specific usage of the system can vary, it is not possible to define all uses of this system. Furthermore, a concrete use-case depends on the content of the knowledge base, the participants and the special cases. That means that the knowledge base needs to contain specific knowledge about the range of the application, special cases must be defined precisely, i.e. the condition that defines if the case has occurred and the action that is triggered afterwards. Both aspects depend on the participants: the *expert* and the *provider*. As already mentioned, the *expert* has a high level expertise. However, the qualification of the *expert* still needs to be verified before he is able to give advice to a *user*. In the ideal case above that describes Steves interaction with the chatbot, it is assumed that the *expert* can handle Steves' problems competently without passing any verification process of his qualification before. To avoid that, the concept needs another human participant: the *provider*. A *provider* is the person who provides a topic to the users and is consequently responsible for it. Furthermore the *provider* has the functionality to validate the competences of an *expert*. After an *expert* registers himself in the mobile application, the system notifies a *provider* over the back-end application. Now, he can verify the *experts* qualifications.

Professional skills are not only necessary for emergency cases. Based on the assumption that the knowledge base of each chatbot is not encompassing, *experts* need to be able to feed the system with new knowledge. This can be done in two different ways which are described as follows:

- **Adding knowledge via mobile application:**
  Similar to an escalation case, the *expert* is notified by the system as soon as the chatbot can not handle the user input or rather does not know the answer. Afterwards, the *professional* needs to pass an automatic procedure to add the answer to the currently existing knowledge base. To add the knowledge to an AI based conversation service like *IBM Watson Assistant* or *Google Dialogflow*, it is necessary to obtain access to further information other than its question and its answer. In addition, the amount of the relevant data may vary depending on the used service. Therefore it is necessary to keep the procedure generic. Furthermore, the mentioned process can be executed in different ways. For

example, it can be presented as a chat, or as a form. Different presentations are also discussed in Chapter 7 and serve as a motivation for further scientific works.

- **Adding knowledge via back-end application:** Besides the fact that the knowledge base should be enlarged whenever the system needs human assistance, the chatbot should be able to be extended manually. Via a server application, a professional can add, update or remove data whenever he or she has the necessary access rights. As described in **FR_23**, the access rights can be assigned per expert and per topic. Besides adapting data, the systems needs to provide features to import or export knowledge. The precise functioning is described in Section 5.3.

  As Figure 5.1 shows, the data which can be edited via the server application is stored in a database. However, the database does not only store the knowledge base but also the user and expert credentials, the possible special cases which can occur, the provided topics and the chat histories per conversation. A *chat history* is used to offer the experts all relevant data. Due to time constraints, the back-end application could not be developed during this master's thesis and needs to get verified in further scientific work. As a temporary solution for the implementation of the back-end application, *Google Firestore* was used to store the necessary data.

As described above, different cases can be handled in different ways. Therefore, it is required to develop a generic solution to handle multiple cases. Section 5.2 describes this in a more accurate way. In general, the concept differs between two types of cases. A *default case* and *special cases*. The second type is already explained above. The first case always occurs, when the system is able to respond properly. In other words, after the user inserts a sentence or question, this input is sent to a conversation service. As shown in Figure 5.1 the system could be connected with services of different third-party providers. As a result of that, the strength of all services can be used. This is also explained more precisely in Chapter 7.

After a registered third-party conversation service, such as *IBM Watson Assistant*, receives the users' input, it replies with a predefined response. However, based on the used algorithms and the size of the knowledge base, the reply can be wrong although it

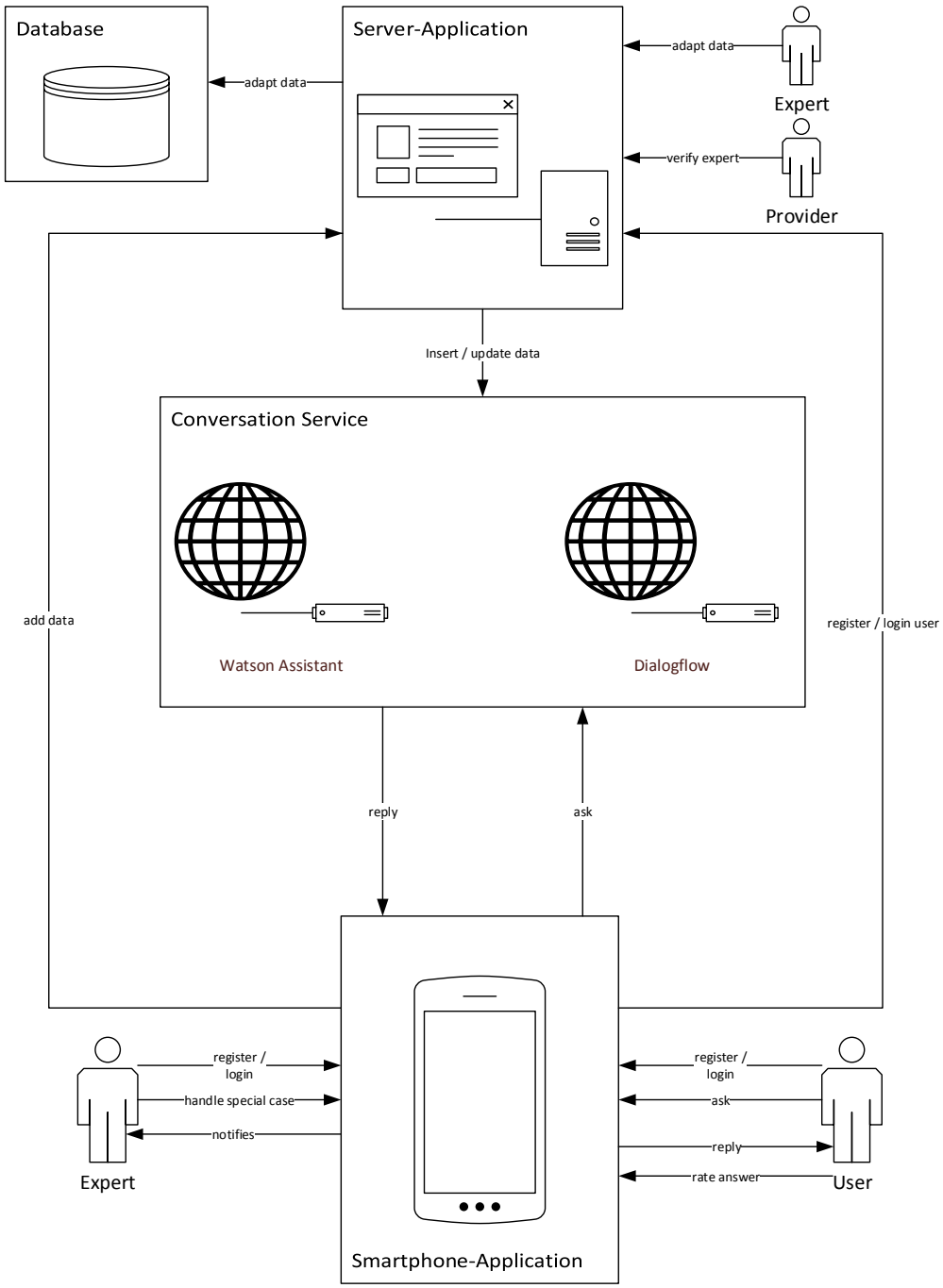


Figure 5.1: Overview of the system process

is categorized correctly by the conversation service. Therefore the user can rate and mark answers as incorrect. The rating feature should be used to improve the accuracy of the responses. Whenever the rating of a system response is below a given limit, it is sent to the experts to get customized. The limit itself can be modified for each topic. The server application is a necessary requirement for this feature. Therefore it could not be implemented completely during this master's thesis.

Another misbehavior can occur, if the conversation service falsely categorizes an unknown input as a known question. For example, *IBM Watson Assistant* returns the answer with the best ranking even if the user input should not be known by the system. Therefore, it can happen that a predefined answer will be returned although the user enters a question without any contextual sense. To counteract this behavior, the user can mark a system response as false. Afterwards the input is unknown for the system and sends a notification to an expert.

To conclude this section it can be said that the concept of the overall system represents a generic approach to support patients and experts in health care but needs to be configured and adjusted first. Without the knowledge about a specific topic, the definition of possible cases that can occur and the support by experts, it can not be able to support the patients. To improve the readers understanding about the single components of the overall system which is represented in Figure 5.1, the following two sections describe the concepts and the architectures of the mobile application and back-end application more precisely.

5.2 Mobile Application

According to Section 5.1, the usage of the mobile application should help people to get a psychological or medical advice. Therefore, each participant needs to create an account first. Section 5.2.1 describes that the registration process of experts and common users needs to be handled in different ways. Furthermore, the same section explains the concept and the graphical sequence of the mobile application.

After giving an overview of the application, Section 5.2.2 illustrates how the interaction between a chatbot service and a user is designed. In addition, it describes the necessary functionality of this process.

The last section clarifies the behavior of the mobile application whenever the chatbot needs human assistance. To support the chatbot optimally, the concept involves both user roles as a human backup.

5.2.1 Concept

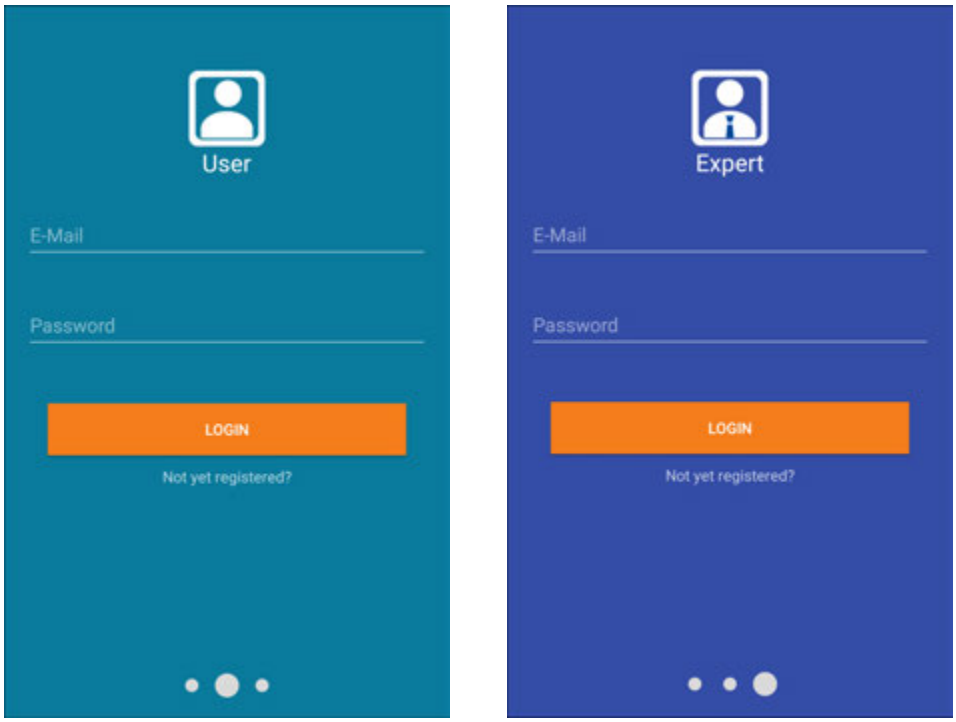
The functionality of the mobile application can be summarized by the three following headings:

- **User Management**
- **Topic Management**
- **Textual interaction with the virtual assistant**

The *User Management* includes the user log-in and the user registration for the two roles: *expert* and *user*. However, the registration process needs to be handled differently for the two participant types. To simplify the usability of the app, the graphical user interface of the structure of the log-in and the registration layout is designed equally. Using the same layout for different usages avoids the development of a second application for the expert but provides the ability to handle the registration separately. The following Figure 5.2 shows the implemented GUI for the user log-in.

To log-in, it is relevant to enter the necessary credentials, *E-Mail* and *Password*. Both need to be specified during the registration process. Once a user or expert is signed in, he is logged in automatically after the app starts until the user signs out manually. Keeping in mind the registration process between the user roles differs, it becomes clear that a division is needed. Whereas a common user does not need any specific qualifications to use the application, it is required that an expert has the necessary qualification in his or her area. Without being a professional in a particular and relevant subject, he or she should neither be able to consult the user nor to extend the knowledge

base of the chatbot. As a countermeasure, the following concept shown in Figure 5.3, illustrates a procedure to avoid this issue. Due to time constraints the process of registering a person as an expert had only been developed theoretically and should be proven in further scientific works.



(A) log-in as user (B) log-in as expert

Figure 5.2: Comparison of log-in screens

As mentioned above, the way how a person should register himself or herself should be consistent. However, as soon as an expert finished the registration process on his or her smartphone, the application notifies the server application about the situation. In addition, the mobile application needs to forward information about the subjects which are covered by the expert. For example, the applicant defines the topics which he can support with. After the server received all necessary data, the responsible providers will be notified. One or multiple providers can start validating the experts qualifications. This can be done by requesting and checking the application documents or by doing an interview. Afterwards, the provider needs to inform the applicant about the application status. The

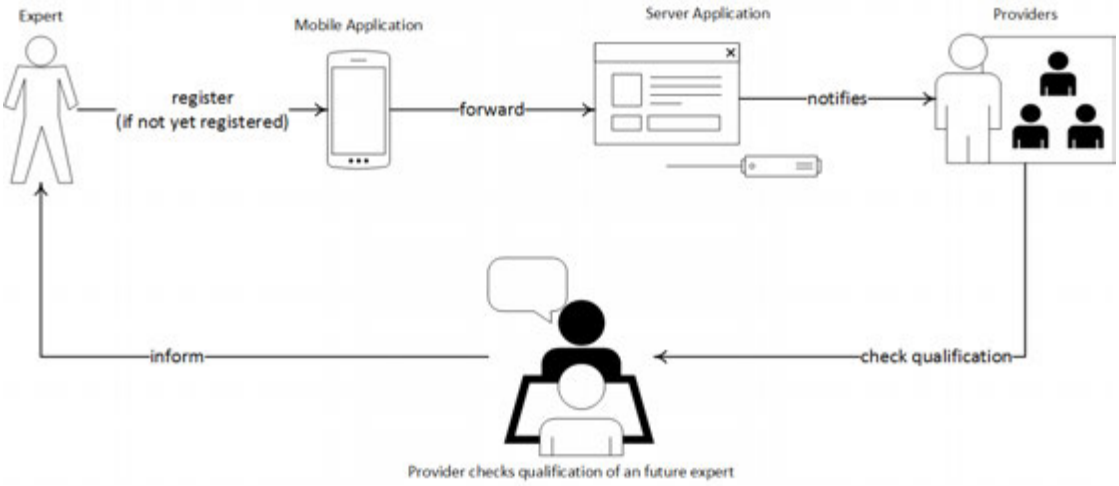
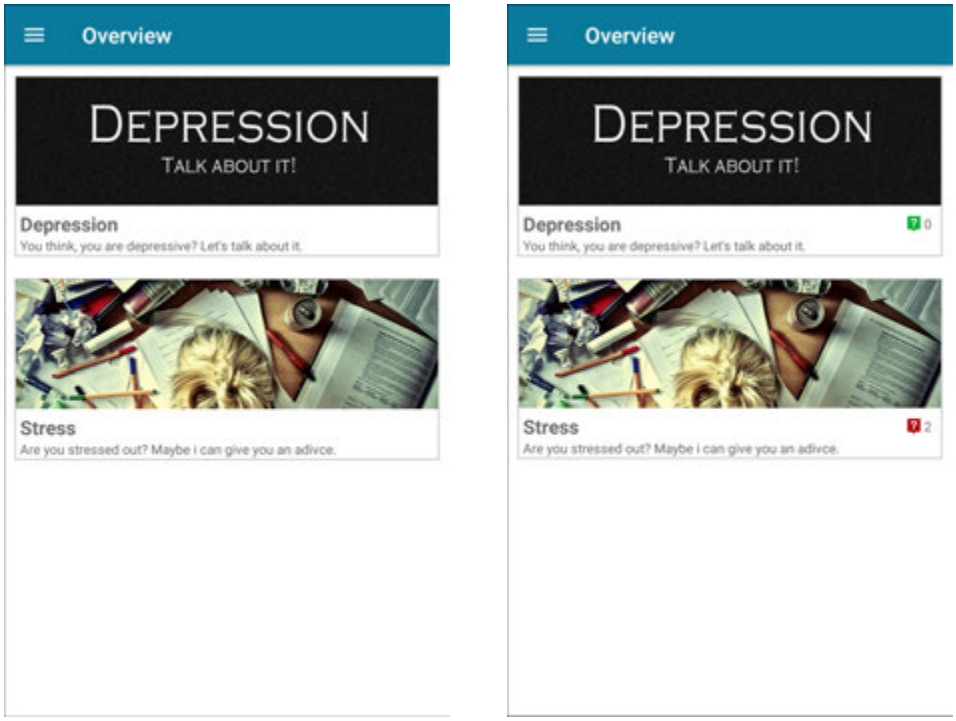


Figure 5.3: Register process of an expert

candidate becomes an expert as soon as he receives an assurance. Professionals can be motivated to apply themselves as an expert by using the collected data. As Section 5.2.3 describes, human assistance can be necessary for e.g. adapting system answers or talking to users directly. Each user has the ability to rate the response of his conversation partner. This rating data can be prepared and used as a reference for the experts.

After the registration process is successfully finished, the user and the expert are signed in and have access to a topic list. As long as a provider has not approved the experts qualification, the expert can only see an empty list. Afterwards, a list is shown to the subjects corresponding their official skills.

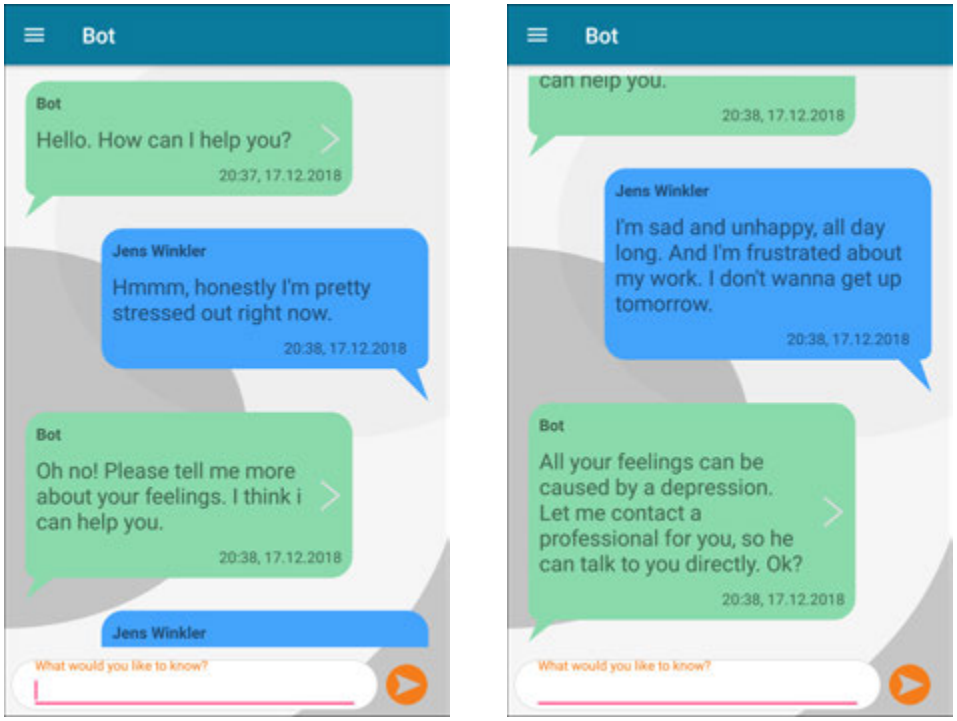
Figure 5.4 displays the different views. The comparison highlights the similarity between the two graphical layouts. However, the functionality is a different one. Showing a topic to an expert with an open request means that the system needs a human backup. This happens e.g. if a user sends a question to the chatbot that can not be handled or an escalation case has occurred. The number of the open cases are displayed next to the red or green highlighted question mark. The whole process is explained more precisely in Section 5.2.3.



(A) Topic-List-View for user (B) Topic-List-View for expert

Figure 5.4: Comparison of Topic-List-View screens

In contrast to the experts' view, if a common user chooses a topic he is forwarded to the virtual assistant. The following Figure 5.5 shows a scenario based on the assumption that a user who is possibly affected by depression clicks on the first list item. After the chat view is shown, a greeting is automatically sent to the chatbot. In this case, the conversation service was configured to respond with a greeting as well. After the first step, the user can talk to the chatbot by entering textual input and clicking the send button. For the communication with a conversation service, the *Android* based framework *Conversation SDK* was developed during this master's thesis. This framework is more accurately explained in Section 5.2.2.



(A) First part of a demo chat. (B) Second part of a demo chat.

Figure 5.5: Chat history of a demo chat.

5.2.2 Conversation SDK

A simple conversation, as it is shown in Figure 5.5, can be realized by the following steps:

1. **Configure a conversation service**
2. **Authorize for the service**
3. **Send requests and handle the responses**

Before a conversation service can be configured, a specific service must be chosen. In terms of the overall system it can be done by the *provider* as he or she is responsible for the provided topic. However, the conversation service which is used during this master's thesis corresponds to *IBM Watson Assistant*. Section 2.5 explains the configuration of this service.

Unlike the first step, the authorization needs to be handled by the application itself. As shown in Figure 5.6 the *Conversation SDK* contains a configuration class which can be used to pass the credentials to a conversation service. However, the structure of the access data can vary per service. To solve this issue the *configuration* is designed as a *JSON Array*. In this way, the configuration can be adapted as needed. The following code snippet demonstrates a configuration structure, which is used for *IBM Watson Assistant*.

```
1 [
2   {"username": "my_username"},
3   {"password": "my_password"},
4   {"versionDate": "yyyy-MM-dd"},
5   {"api_endpoint": "https://my.endpoint.com"},
6   {"workspace_id": "my_workspace_id"}
7 ]
```

Listing 5.1: *JSON* Example of the *IBM Watson Assistant* configuration

The *username* and *password* are generated per service. This means that the usage of multiple services would lead to multiple *usernames* and *passwords*. As the configuration is *JSON* formatted, an adjustment like adding another *username* with a different identifier can be performed simply. The *versionDate*, *api_endpoint* and the *workspace_id* are defined per *Skill* as it cannot be ruled out that different *Skills* can be accessed. The same procedure as mentioned above can be used to extend the configuration about e.g. multiple *workspace_ids*.

After a *configuration* is defined for a given conversation service, the mobile application can send requests via an *API* and handle the received responses. This can be performed by using the *Conversation SDK*.

The *Android* based framework is designed to unify various third-party chatbot services allowing a certain flexibility. Furthermore, this approach enables to make usage of the specific strength of each service. The fact that multiple third-party *Software Development Kits* (SDK) can differ from each other must be handled by the framework.

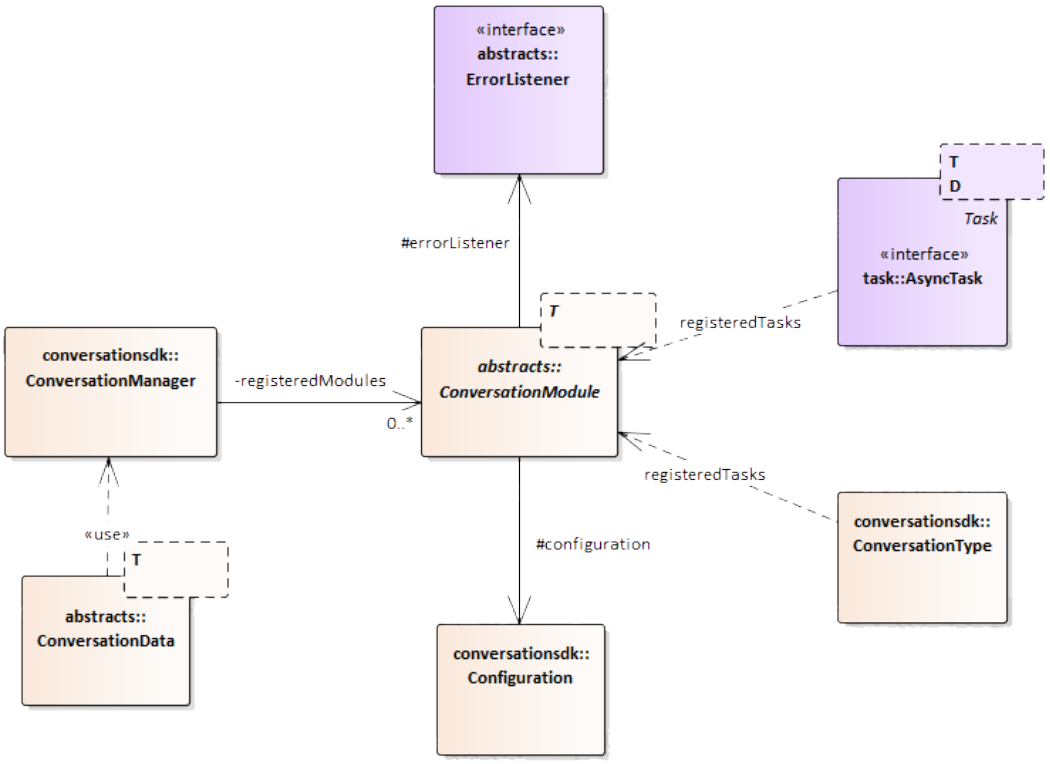


Figure 5.6: System architecture of *Conversation SDK*

The general approach is that each third-party service can be wrapped by a *ConversationModule*. To make use of a module, it can be registered in the *ConversationManager*. This class can be used to perform the specific functionality of the registered modules. As many third-party conversation services provide *REST* APIs, the scope of the *ConversationManager* can be reduced to the followings functions:

- **CREATE** - Create a new data element in the knowledge base of the service.
- **GET** - Get a single or multiple data elements from the service.
- **UPDATE** - Update one or multiple data elements.
- **DELETE** - Remove one or multiple data elements.
- **ASK** - Send an input to the service and receive a response.

The reason of creating the *ConversationManager* was to keep the single *API* calls simple. As an example, if a user input should be sent to a conversation service, it is sufficient to call the function, shown in Listing 5.2.

Executing this function means to execute a task with the given data for each registered *ConversationModule*. The type of the operation is defined by passing the constant *TYPE_OPERATION_ASK*. The usage of types per functionality keeps the framework generic. That means, each module can contain specific methods but can be handled in the same way by calling *execute*.

```
1  /**
2   * Sends the given {@link ConversationData} to all
3   * registered {@link ConversationModule}s and
4   * waits for its result.
5   *
6   * @param data The specific data must contain all
7   * necessary information for the execution.
8   * @param callback returns the response.
9   */
10 public void ask(ConversationData data, TaskCallback callback) {
11         for(ConversationModule module :
12                     this.registeredModules.values()){
13           module.execute(Constants.TYPE_OPERATION_ASK, data
14                   ,callback);
15         }
16 }
```

Listing 5.2: Function *ask* of the *ConversationManager*

However, a *ConversationModule* is only able to execute an operation, if it contains tasks. Due to time constraints, a module can only execute *AsyncTask*s. An *AsyncTask* represents a functionality which can be executed asynchronously. Synchronous tasks

are currently not supported using the base methods listed above. To make a task work, it must be implemented first. This approach allows to expand the *Conversation SDK* by adding and implementing new tasks and modules and is explained more accurately in Section 6.1.

To register a task to a *ConversationModule* it must be assigned with a *ConversationType*. This class represents a combination of an *OPERATION_TYPE* and a *DATA_TYPE*. An *OPERATION_TYPE* represents a unique identifier which refers to a specific operation. As shown in Listing 5.2, using the type *TYPE_OPERATION_ASK* means to send a request to a *conversation service*. The *DATA_TYPE* reflects the type of a data element which is supported by the conversation service. Referring to *IBM Watson Assistant*, a possible data type can be an *Intent*, *Example* or *Entity*. The *ConversationType* combines both mentioned types allowing software developers to create and to adapt a module in the way they like. The table below demonstrates possible examples of *ConversationTypes* and what they can be used for.

Data / **Operation**	**INTENT**	**EXAMPLE**	**ENTITY**
CREATE	Create Intent	Create Example	Create Entity
GET	Get Intent	Get Example	Get Entity
DELETE	Delete Intent	Delete Example	Delete Entity

Table 5.1: *ConversationTypes* as a result of the combination of *DATA_TYPE*s &*OPERATION_TYPE*s

As already mentioned, the structure of third-party conversation services can vary. That means, to unify multiple third-party libraries the framework needs to allow the definition of data elements and operations per service. This issue can be solved by using the *ConversationType*.

After a module contains tasks, they can be executed. As Listing 5.3 shows, the method expects besides an *operationType*, a *ConversationData* and a *TaskCallback*. The passed data element contains all necessary information which is used by the registered task. Besides, it contains the *DATA_TYPE* which is combined with the given *OPERATION_TYPE* to find the registered *AsyncTask*.

```
1 public void execute(String operationType, ConversationData data,
2                         TaskCallback callback){
3    ConversationType ct = new ConversationType(operationType,
4                 data.getType());
5    AsyncTask task = this.registeredTasks.get(conversationType);
6    task.execute(this, data, callback);
7 }
```

Listing 5.3: Function *execute* of the *ConversationModule*

As soon as the task with the *ConversationType* is found, the *ConversationModule* executes the task. Whenever an operation was executed successfully, the *TaskCallback* which is included in the *Conversation SDK* can be used to receive a calculated response. The responses of different tasks in different *ConversationModule*s are returned by the same *TaskCallback*. To catch errors, the developer can make use of the *ErrorListener*.

The presented framework is demonstrated with precise examples in Chapter 6. In addition to the mentioned functionality, the *Conversation SDK* also contains features to run multiple tasks sequentially. This is explained in the following section.

5.2.3 Human Backup

Chatbots still need human assistance due to the fact that they are not omniscient. In fact, the quality of a chatbots' response depends on the used algorithms and the knowledge base. Like human beings, the system can only understand a question and reply to it, if it has already heard it before. Otherwise the chatbot needs to learn, i.e. it needs to implement new knowledge.

As the application is divided into two different user roles, it seems reasonable to use both groups to support the chatbot whenever it needs assistance. This support is explained in the following sections.

Human Backup by Users

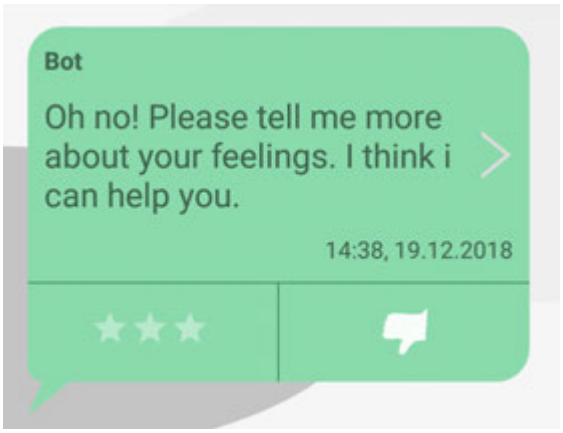


Figure 5.7: Speech Bubble of the chatbot

Although it is not required that a user needs specific qualifications, he can still help to improve the system. Therefore, each speech bubble displayed to the user provides a functionality to rate system responses from 0 to 100 percent. To rate a reply, the user can slide over the *Rating Bar* which is shown in Figure 5.7. If none of the stars are highlighted it means that the user totally disagrees with the response. If all of the stars are highlighted, the user agrees to this answer to 100 percent. However, the rating view needs to be touched once, to rate the response at all. This should avoid to distort the assessment by ignoring the rating system. As soon as an output is rated, the average rate is updated. Whenever this value gets below a predefined limit, the output is sent to the experts. Afterwards, they can adjust the response and update the knowledge base. This procedure can be done in multiple cycles.

Due to time constraint, this feature could not be implemented completely during this master's thesis. The reason for that is the need of the back-end application. Therefore, the theoretical aspect should to be proven in further scientific work.

Besides the rating system, the user can correct the system whenever it gives a response that does not reflect the input. As a side effect of a probability calculation, a chatbot can recognize an input that is not yet defined in the knowledge base. To counteract this behavior, the user can mark the response as unknown. Afterwards the system would reply with a fallback answer if the same question appears again.

Human Backup by Experts

In contrast to common users, experts are responsible for the knowledge of the chatbot. The experts need to teach the chatbot everything about a specific topic, otherwise the system would not be able to give useful responses. Additionally, experts are also needed to handle critical cases. Such cases can occur if the system recognizes a medical emergency or the user talks about suicide. As it is not possible to cover all possible situations, the system provides functions to add, remove or modify cases. To be more precise, the usage of inheritance allows the developer to create custom cases which can be registered to the *ConversationManager*. This is shown in Figure 5.8.

The class *ConversationCaseManager* is responsible to manage all existing cases. This includes storing the predefined *ConversationCases* temporarily in a list and getting them back to check if a special case occurred. Before the verification of a case is explained, it is relevant to understand the meaning of a *ConversationCase*. The class *ConversationCase* represents a random situation that can occur. This includes a special case such as the detection of suicidal thoughts, as well as the input of a normal question from the user. Each possible situation must be predefined and can be loaded from the server. This will be explained more precisely in Section 6.3. However, because each case can vary, they need to be handled separately. That means that a normal exchange between the user and the chatbot is dealt in a different way than a medical emergency. As shown in Figure 5.8 the system, which was developed during this master's thesis can manage the following cases: *DefaultCase*, *SuicideEscalation* and *UnknownInputCase*. A *DefaultCase* presents the situation where the system is able to understand the users' input and answers properly. The case *SuicideEscalation* occurs if suicidal thoughts are detected by the system after the user inserts an input. The last listed case, the

UnknownInputCase, represents a situation where the user entered an input that is unknown to the chatbot.

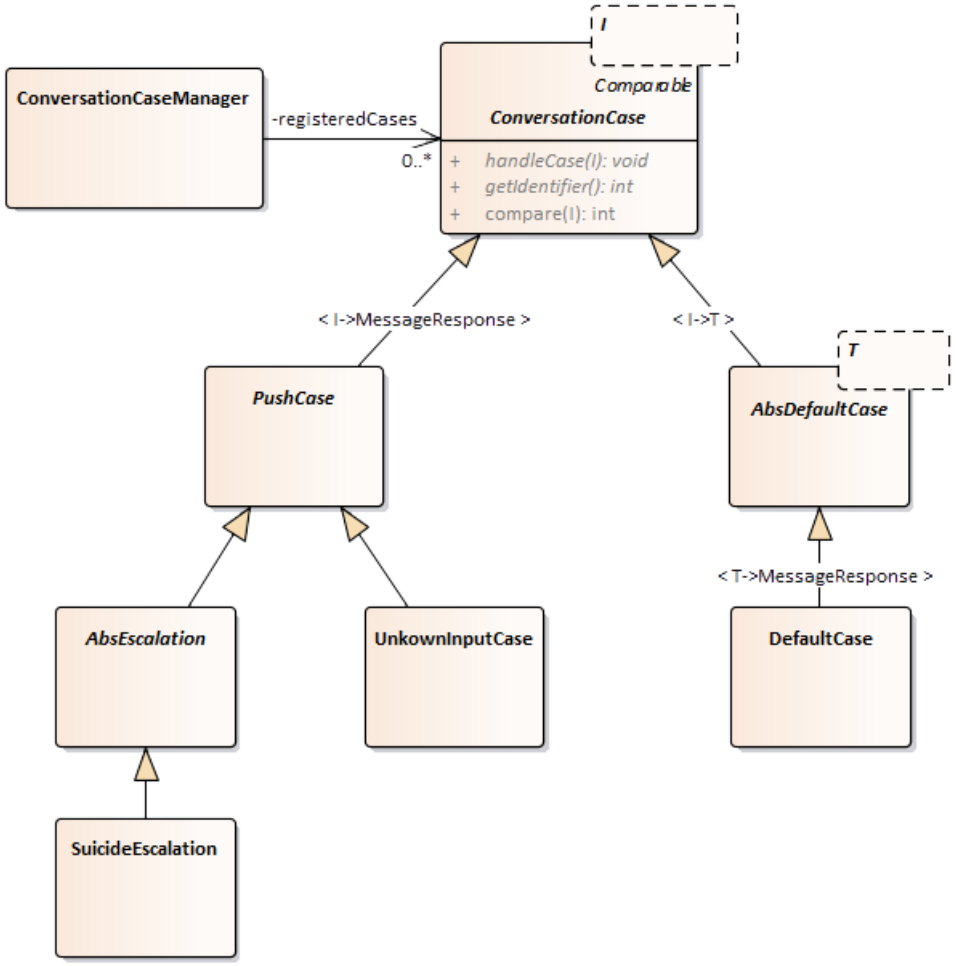


Figure 5.8: *ConversationCase* architecture used in this master's thesis

As already mentioned, to create further concrete special cases, the developer must extend the class *ConversationCase* and implement the methods *getIdentifier*, *compareTo* and *handleCase*. These functions are necessary to verify which case has occurred and how it should be handled. The whole procedure describing how the mentioned cases are handled is shown in the following Figure 5.9.

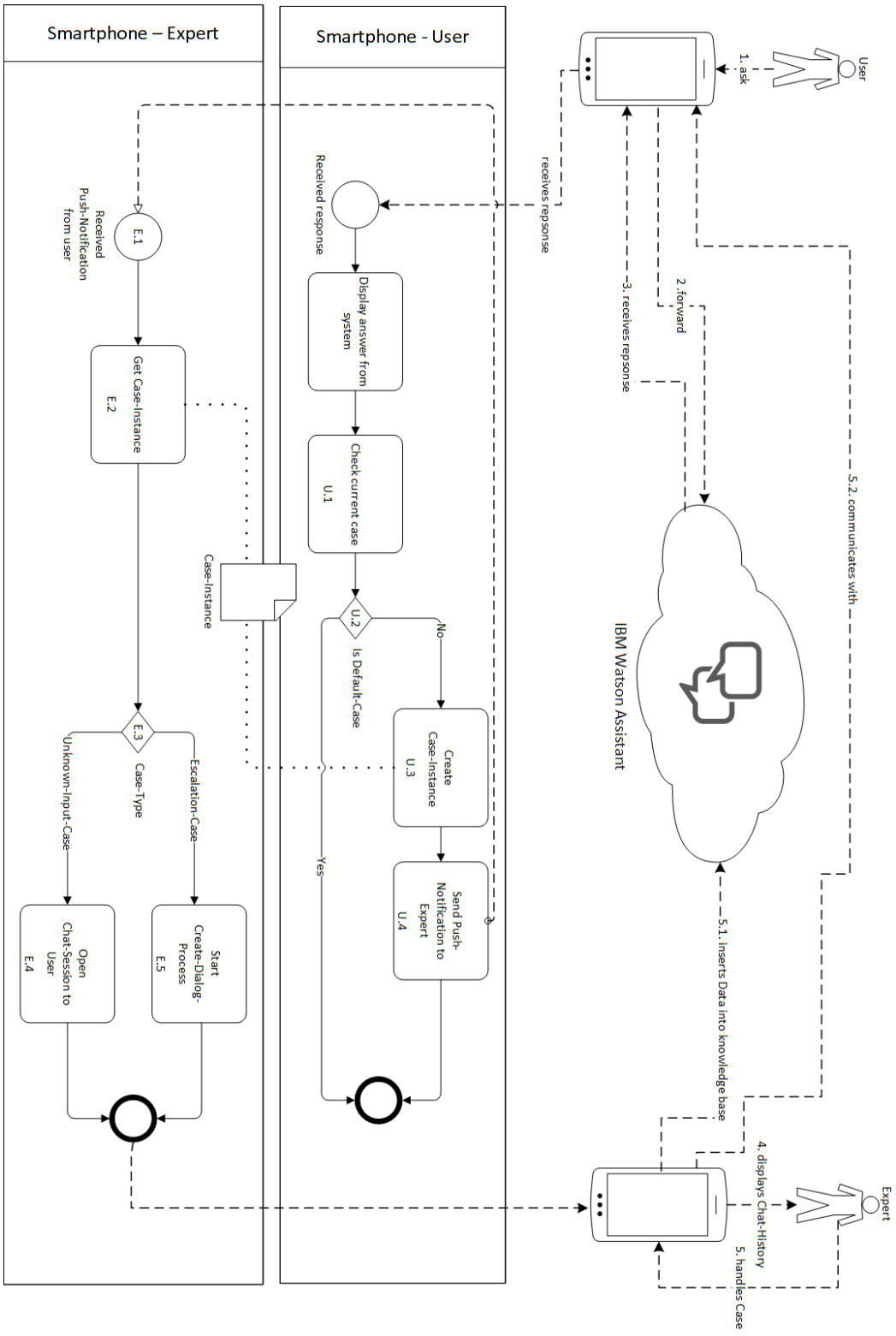


Figure 5.9: *Procedure* of handling different cases

The whole procedure starts with a users' input. As soon as the user enters an input (Step 1, Figure 5.9) and sends it to the chatbot (Step 2, Figure 5.9), he receives a response (Step 3, Figure 5.9). This response is used to determine the current case (Step U.1, Figure 5.9). To check the current case, the *ConversationCaseManager* runs through all registered *ConversationCases* and calls their *compareTo* method. This function determines if the response of the chatbot can be associated with a case. As an example, the function *compareTo* of the *ConversationCase SuicideEscalation* verifies if the users' input contains possible statements that can be associated with suicidal thoughts. Therefore, the function checks whether the response of the chatbot includes the *Entity* "suicide". This *Entity* contains possible statements like "kill myself" or "I want to die". Whenever the mentioned *Entity* is detected, the current occurred case is identified as a *SuicideEscalation*. Section 6.3 demonstrates a sample to further improve the readers' understanding of a possible implementation.

As each case has its own implementation of the function *compareTo*, different cases can be distinguished. This is necessary to handle situations differently. Figure 5.9 shows that the system distinguishes between a *DefaultCase* and special cases (Step U.2, Figure 5.9). When the *DefaultCase* occurs, the mobile application displays the chatbots response to the user. However, if a special case is found the application creates a *CaseInstance* (Step U.3, Figure 5.9). This class represents the occurrence of a special situation and contains all necessary information that is needed to manage the case. At Step U.3, the system found a special case that needs to be handled. Therefore the *handleCase* function that is part of the class *ConversationCase* is called. The implementation of this method reflects the action that is executed to handle the case. As the *SuicideEscalation* or *UnknownInputCase* inherit from *PushCase*, the situation is handled by sending a *Push Notification* to all experts who are responsible for the topic (Step U.4, Figure 5.9). That means that a clickable notification appears on devices of all experts responsible for the certain topic. If an expert clicks on the notification, the mobile application starts (Step E.1, Figure 5.9). The received *Push Notification* contains the identifier of the created *CaseInstance*. Consequently, the mobile application on expert side can load the *CaseInstance* from the back-end (Step E.2, Figure 5.9). This data element is used to determine the type of the occurred case. Depending on

the *Case-Type*, the application decides how the situation should be handled (Step E.3, Figure 5.9). While the detection of suicidal thoughts leads to a direct communication channel (Step E.4, Figure 5.9) between the *expert* and the *user*, the detection of an unknown user input creates a process that needs to be passed through the *expert* and aims to the addition of the unknown input into the knowledge base (Step E.5, Figure 5.9). However, before the expert can step in, the application always displays the chat history to the expert after the app started (Step 4, Figure 5.9). This is necessary to get an impression of what happened.

As already mentioned, the first case creates a communication channel between the user and the expert. Afterwards the expert can offer his assistance (Step 5.2, Figure 5.9). A textual conversation between the expert and the patient presents only one possibility of handling a critical case like the detection of suicidal thoughts. Further approaches can be the call of a psychologist or making an appointment in a medical center. Therefore, the implementation of the function *handleCase* needs to be changed.

The second case (Step E.5, Figure 5.9) leads the expert into an automatic procedure. During this process, the expert is asked about the necessary data, so the unknown input can be successfully added to the chatbots knowledge base (Step 5.1, Figure 5.9). This procedure is shown in Figure 5.10 and demonstrates a possibility how knowledge can be added to a chatbot using a dialog. To test the process it was implemented during this master's thesis using *IBM Watson Assistant.* However, it should kept in mind, that the process can vary depending on the used chatbot service. Accordingly, the process can differ in its content and its presentation form. As a result of that, it is necessary that the procedure can be modified and adapted to its requirements. To realize this requirement, a *Taskqueue* was developed during this master's thesis. This system component allows the application developer to build a customized process that includes multiple steps. In contrast to the *ConversationManager* that calls its functions asynchronously, the *TaskQueue* calls each task synchronously. A reason for this behavior is the fact that single tasks may depend on other tasks. As an example, the tasks *Add Example to Intent* and *Create new Intent* in Figure 5.10 need the result of their previous task *Wait for user input* before it can be executed. To simplify the usage of the *TaskQueue*, it can be executed by the *ConversationManager*. However, the *TaskQueue* represents only an

abstract component that does not contain any tasks in the beginning. Therefore, new tasks must be implemented and added to the *TaskQueue* before it can be executed. The usage of this component and its tasks is explained more precisely in Section 6.2.

As mentioned, the presentation form of the procedure which was implemented during this master's thesis is a chat view. During a guided conversation, the expert can interact with the system by entering his answers. The goal of the whole process is to integrate the unknown user input, so the chatbot can reply to it afterwards. To make the chatbot able to reply, the users question must be added to the knowledge base. Referring to *IBM Watson Assistant*, adding knowledge means to add an *Example* to an *Intent*. If the *Intent* does not already exist (Step S3, Figure 5.10), it must be created first. Therefore the expert is asked to categorize the unknown user input and the already existing *Intents* are displayed to the expert (Step S1, Figure 5.10). If he or she inserts the name (Step S2, Figure 5.10) of an existing *Intent*, the unknown input is added to it as an *Example* (Step S5, Figure 5.10). Otherwise a new *Intent* with a new *Example* is created (Step S4, Figure 5.10).

Afterwards, the expert can add a new *Entity*. Therefore, the application displays, similar to the *Intents*, the existing *Entities* with their *Synonyms* to the expert (Step S6, Figure 5.10). Afterwards, the expert can create (Step S8, Figure 5.10) or extend (Step S9, Figure 5.10) an *Entity*. However, adding an *Entity* is optional. That means the expert can skip this step by entering a keyword, such as "Skip" (Step S7, Figure 5.10).

The last part of this procedure is to add an answer to the user input. Therefore, the mobile application shows an instruction text to the *expert* (Step S10, Figure 5.10). Afterwards he can insert the answer that refers to the unknown user input (Step S11, Figure 5.10). To complete the process, the app still needs to know what should be done after the validation of the new inserted node. Therefore, the expert can insert the name of a displayed neighboring node (Step S12, Figure 5.10). After that, the new node will behave in the same way the neighboring node does.

If the expert finishes the process successfully, new knowledge is added to the chatbots' knowledge base (Step S13, Figure 5.10). To avoid adding the same question and answer

for multiple services, it is necessary to manage the data in a central knowledge base. The following section illustrates the theoretical aspects of this approach.

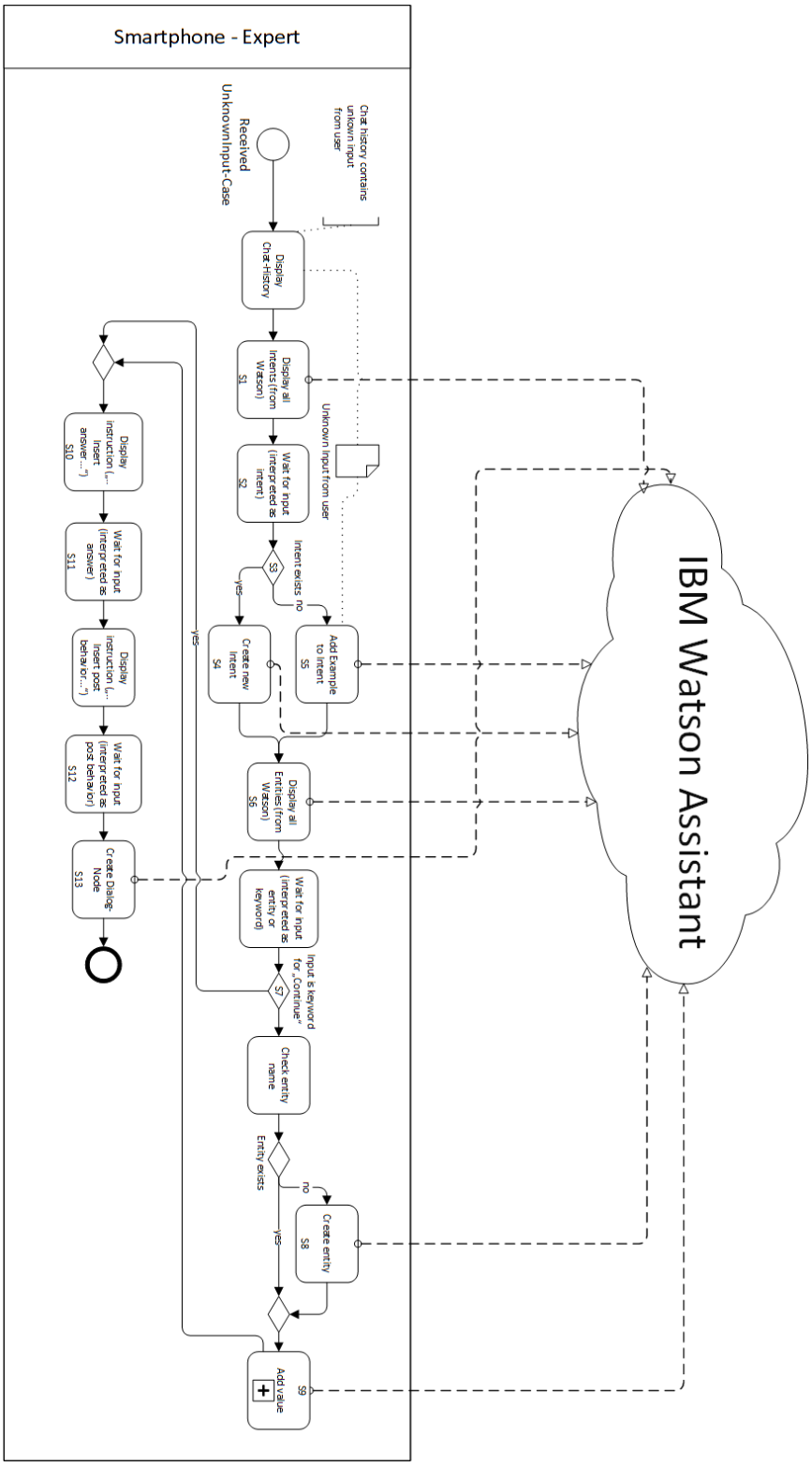


Figure 5.10: Adding new knowlege to *IBM Watson Assistant*

5.3 Back-End Application

As already mentioned, the back-end application can be used to manage and edit the knowledge of a chatbot. As the overall system should be able to support multiple chatbot services, it is required to have a central knowledge base that stores all data elements for all topics. The following Figure 5.11 demonstrates this approach and the interaction between the knowledge base and multiple chatbots.

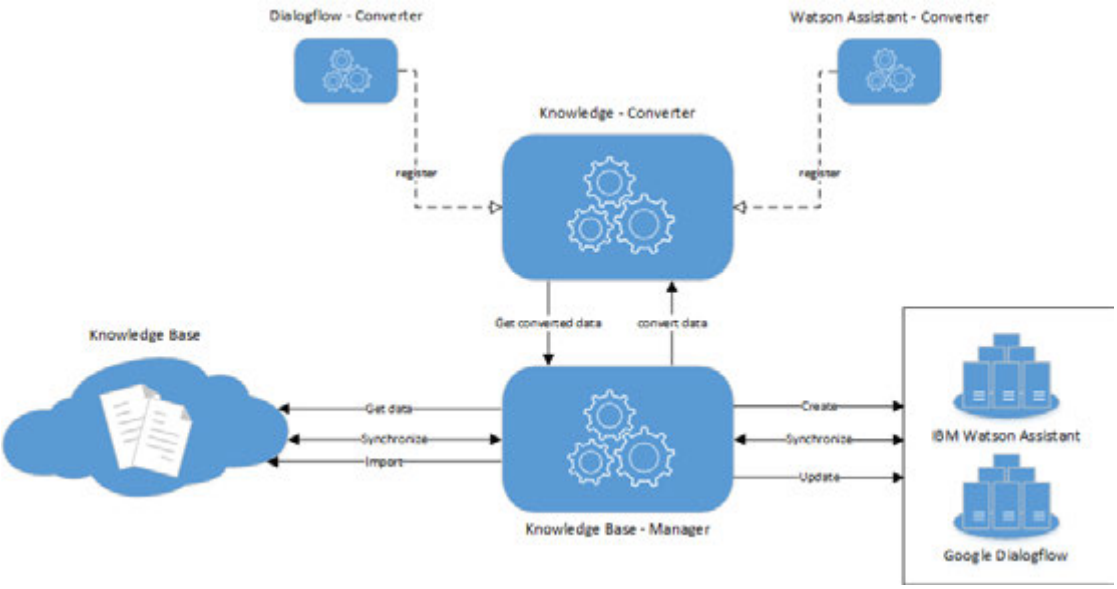


Figure 5.11: Interaction between chatbot services and the central knowledge base

To summarize the procedure, the back-end application should be able to synchronize the data of the central knowledge base with all connected chatbots. As soon as the knowledge base changed, the synchronization is triggered by the *Knowledge Base - Manager*. This program gets the data from the database, converts it and updates the individual chatbots. The conversion of the data elements is required as each chatbot can contain its own data structure.

To keep the process generic, the *Knowledge Base - Builder* should be expandable by adding *Converters*. A *Converter* represents a program which expects the data structure of the knowledge base and converts it into a specific data structure for a chatbot. The

conclusion of the definition is that each chatbot in use needs a specific *Converter* to be adapted or extended.

After the *Knowledge Base - Builder* gets the customized data from the *Converter*, the external knowledge base can be created or updated. Creating a knowledge base is necessary whenever a new conversation service is used. As it can not be assumed that a new chatbot already contains knowledge, the overall system must provide a functionality to pass the data elements of the central knowledge base into the chatbots knowledge base. Afterwards it can be updated, as soon as the knowledge changed. The use of one knowledge base of multiple chatbots and their different data structure could lead to functional limitations. To avoid that, the central knowledge base as well as the synchronization process needs to be adaptable. This theoretical approach is described in the following sections.

5.3.1 Knowledge Base

Due to time constraint, the structure of a central knowledge base could only be designed theoretically. The goal of this general approach is to avoid limitations of the specific chatbots. As an example, *IBM Watson Assistant* contains *Intents*, *Entities* and *Dialog-Nodes* as dialog components, whereas *Google Dialogflow* contents only *Intents* and *Entities*. Besides the different number of dialog components, the data structures of an *Intent* and *Entity* vary although they have the same name. That shows that data elements of different chatbots could be similar but not equal. Furthermore it should be assumed that dialog components can bex totally different. This fact makes it difficult to generalize them. The approach of generalizing the components would bring the advantage of storing all data in one single database, which represents the central knowledge base. Consequently, it would be easier to manage the data. However, the predefined attributes must cover all possible properties of the existing third-party knowledge base components. Otherwise adding a new conversation service could lead to a total refactoring process as the present data structure does not fit with the central knowledge base. Without an adaption, it could lead to a lack of information and the limitation of functionality.

To avoid the discussed issue, the central knowledge base is split into two different sections which are shown in Figure 5.12.

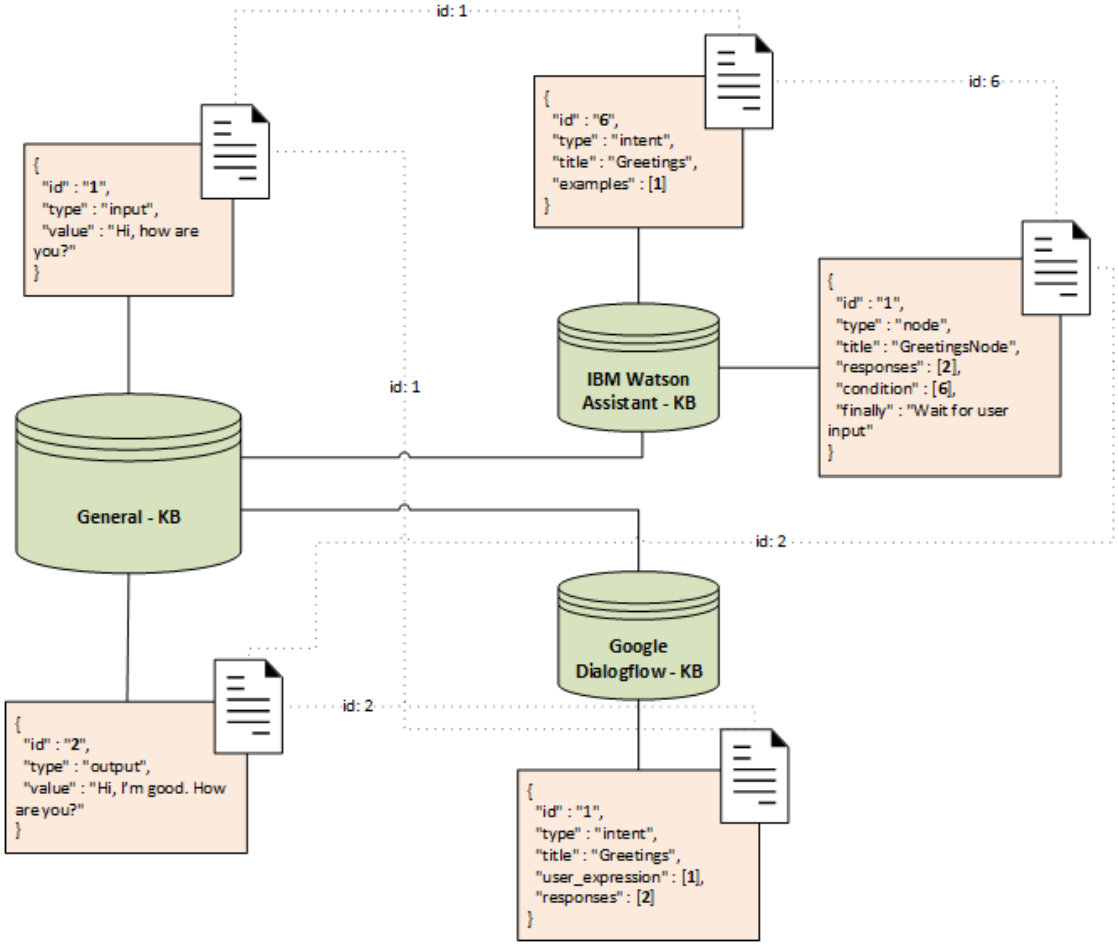


Figure 5.12: Structure of the central knowledge base

According to the definition of a chatbot in Section 2.1, each chatbot has the same work-flow. A system expects a natural language input and responds with a meaningful answer. Considering the principle in a superficial way can lead to the conclusion that a chatbot only needs two different data type for its knowledge base: *input* and *output*.

The mentioned data types are located in the first section which is represented by the *General Knowledge Base*. An *input* generalizes all user requests. It represents a

question as well as a statement. The output can be considered as the system response. As the *General Knowledge Base* only contains these two data types, the second section is responsible for the specific chatbot components.

Figure 5.12 shows the *IBM Watson Assistant - KB* and the *Google Dialogflow - KB*. Both databases can be defined as a *Specific Knowledge Base*. A *Specific Knowledge Base* can be used to map the structure of a chatbots' knowledge base. Because chatbot services are constantly being developed, managing the structure of each knowledge base can bring a benefit. Whenever a chatbot is evolving, the associated *Specific Knowledge Base* can be adapted easily. On the other side, a negative aspect of this concept is the possible redundancy of data elements. As an example, an *Intent* must exist for *IBM Watson Assistant* as well as *Google Dialogflow* if both services are supposed to be usable. The input and the output form the foundation of the knowledge and are stored in the *General Knowledge Base*. Redundancy only exists in specific knowledge bases due to similar processes in the different chatbots.

As the two mentioned sections need to work together, so it can be combined as a single knowledge base, the data elements can be connected. Figure 5.12 shows the general principle. Database entries of a *Specific Knowledge Base* can refer to an *input* or an *output*. That means, that e.g. an *Intent* of *IBM Watson Assistant* has an *Example* which corresponds to the *input* with id = 1, whereas the *Intent* of *Google Dialogflow* has the same *input* but as a *User Expression*.

This connection is necessary to create the complete specific data structure, which is required to synchronize the chatbots' knowledge base. The process of the synchronization is described in the following section.

5.3.2 Synchronization

Considering the synchronization, the two following aspects should be regarded:

- Synchronization between the *General Knowledge Base* and a *Specific Knowledge Base*
- Synchronization between different *Specific Knowledge Bases*

Whenever the content of the central knowledge base changes, the *General Knowledge Base*, the *Specific Knowledge Base* or both databases must be edited. If the expert e.g. adapts an answer, it is not absolutely necessary to edit a *Specific Knowledge Base*. The reason of the revision could be an inaccurate wording or spelling. Therefore, the structure of the dialog can be kept the same and it is sufficient to edit only the value of the associated *output*. Similar to this case, the adaption of that data element would be enough if an *Intent* in the specific knowledge base contains a wrong *input*. It could be deleted and the *General Knowledge Base* does not need to be reworked. However, as soon as the *General Knowledge Base* is extended, both parts must be adapted. It could be possible to simply adapt the references which are shown in Figure 5.12 or to add new specific data elements to the *Specific Knowledge Bases*. The described approaches refer to the first aspect which is listed above.

As all components of the central knowledge base should behave like one single knowledge base, it is also necessary that the *Specific Knowledge Bases* synchronize to another. That behavior can be illustrated by the following scenario.

A user enters a question and sends it to a chatbot, which was created by the usage of *IBM Watson Assistant*. The interaction with the chatbot is performed via a mobile application. The chatbot does not know the input and asks for human assistance. The expert passes the process which is described in 5.2.3 and the *input*, an *output* and a new *Intent* is added to knowledge base. In addition, the *Example* of the *Intent* stored in a *Specific Knowledge Base* corresponds to the new *input*. A few minutes later, another user want to know the answer of the question which was recently added to the knowledge base. However, she asks the chatbot by talking to *Google Home* which has enabled the mentioned skill. Without a synchronization between the different data structures the virtual assistant can not reply properly because his knowledge base was not updated.

Under the conceptualized approach to synchronize the two *Specific Knowledge Bases*, the mentioned issue can be avoided. However, it is necessary to have information about the mapping between both structures. Therefore it is required to create a mapping scheme for all structure dependencies. An example of a possible mapping scheme is shown in Figure 5.13.

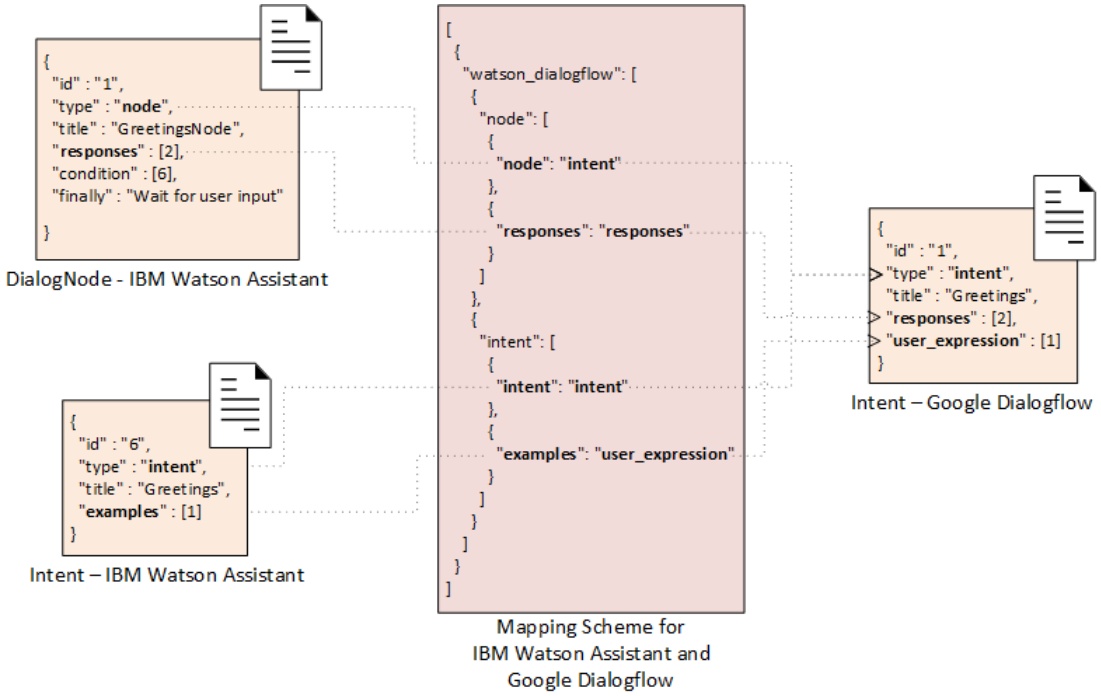


Figure 5.13: Example of a possible mapping scheme for *IBM Watson Assistant* and *Google Dialogflow*

It should be kept in mind that the explained concept of the server application and its knowledge base could not be proven during this master's thesis due to time constraints. Therefore it represents a theoretical approach for further scientific works. In the contrary, the explained components of the mobile application were implemented during this thesis. The precise use of selected aspects are illustrated in following Chapter 6.

6

Implementation Aspects

This chapter demonstrates three important system components of the mobile application which are explained in Chapter 5. To be more specific, it contains the *Conversation SDK*, the *TaskQueue* and the *ConversationCase*. The aim of this chapter is to improve the readers' understanding of their usage as an application developer. This is necessary to extend or adjust the system components and is explained through the presentation of code snippets of implemented classes in the following sections.

6.1 Conversation SDK

Section 5.2.2 already described the main reason and the use of the *Conversation SDK*. However, this section illustrates how it can be extended by a new module. Therefore a *ConversationModule* must be created and new tasks have to be implemented and added to the module.

```
1 class IbmWatsonModule extends ConversationModule<Conversation>
```

Listing 6.1: *IBMWatsonModule*

Creating a new module means to extend the *ConversationModule*. The generic type of the *ConversationModule* describes the type of its *Util*. The module which is shown in Listing 6.1, wraps the *Watson APIs Java SDK* and its *Util* corresponds to a *Conversation*-Object. This object can be passed to the implemented tasks as it is required to execute

the API call. What kind of API call will be executed depends on the task. An example is demonstrated in Listing A.1.

Before a specific *AsyncTask* is created, the *IBMWatsonModule* needs to be configured. That means that the *Configuration* of the module must be loaded. The necessary information can be requested from the *Server Application* to keep the overall system flexible. After the application received the *Configuration*, it can be used to initialize the *Util*.

```
1    @Override
2    public void init(Context context) {
3        configure();
4        initUtil();
5        registerTasks();
6    }
```

Listing 6.2: Initialization of the *IBMWatsonModule*

The initialization itself depends on the used third-party library. Using the *Watson API Java SDK* requires a *versionDate*, an *endpoint* and the user credentials. The single attributes are already explained in Section 5.2.2.

```
1    private void initUtil() {
2        service = new Conversation(configuration.attributes.get
             ("versionDate"));
3        service.setEndPoint(configuration.attributes.get("
             api_endpoint"));
4        service.setUsernameAndPassword(configuration.attributes
             .get("username"), configuration
5                .attributes.get("password"));
6    }
```

Listing 6.3: Initialization of the *IBMWatsonModule*-Util

The mentioned functionality can be defined in the *init* method, which is shown in Listing 6.2. To call this function, the module should be registered in the *ConversationManager* first. This class contains the method *initRegisteredModules* which loops over all registered *ConversationModule*s and calls their *init* function. This procedure is shown below.

```
1        public void initRegisteredModules(Context context) {
2                for(ConversationModule module :
                    registeredModules.values()){
3                        module.init(context);
4                }
5        }
```

Listing 6.4: method *initRegisteredModules* of *ConversationManager*

Creating a new module only makes sense if it contains a required functionality. Therefore, it is necessary to add tasks to a module before the initialization is triggered. This part explains the creation of a specific task as the general principle is already explained in Section 5.2.2. To illustrate the procedure, a *DeleteIntentTask* that is shown in Listing A.1 is used. This task can be used to delete an *Intent* of a *IBM Watson Assistant* chatbot. To execute the procedure successfully, the *Util* and the relevant information are required. For accessing the *Util*, the *execute*-function expects the *IbmWatsonModule* as a parameter. The specific type of the module can be defined as a bounded type parameter. Before deleting an *Intent* from the chatbot, it is necessary to pass the affected *workspace* and the *Intent*. To solve this issue, the *Watson API Java SDK* provides a *DeleteIntentOptions*-Object. However, this object needs to be filled with information. Using the *Conversation SDK* it is required to pass this data wrapped as a *ConversationData*-Object, when calling a tasks *execute*-function. To access the specific values, predefined identifiers can be used. More accurately said, to access the *Workspace-ID*, the identifier *KEY_WORKSPACE_ID* can be used. After all necessary data is passed to the *Util*, the API request can be sent to the chatbot. The response can be returned by using a *TaskCallback*. Therefore it is necessary to call the *onCallback*-function as

soon as the response is returned. Similar to the respsonse, the *IbmWatsonModules*' *ErrorListener* should be triggered after an error is catched.

As described in Section 5.2.2, a *ConversationModule* can only handle *AsyncTasks*. However, the usage of a *TaskQueue* allows the *ConversationManager* to execute *SyncTasks* as well. This approach is illustrated in the following section.

6.2 TaskQueue

Figure 5.10 in Chapter 5 describes a gradual procedure to add new data to a knowledge base. Due to the possible modifications of this process structure, it is necessary to adjust it without an adaption of the system architecture. Therefore, the *TaskQueue* was developed during this master's thesis. This class can store single tasks which are called *TaskQueueItem* and executes them step by step. This generic approach offers the advantage that it is possible to create a completely new procedure by combining multiple tasks which can be implemented individually.

Combining multiple tasks means to create an execution order and to build dependencies between the single tasks. Registering one *SyncTask* before a second one causes the first task to be executed before, too. Therefore, the execution order can be determined by registering *SyncTask*s one by one in the *TaskQueue*. To create dependencies, the following predefined flags can be used.

```
1        public static final int NONE = 995;
2        public static final int PREVIOUS_ITEM_SINGLE = 997;
3        public static final int PREVIOUS_ITEM_MULTIPLE = 996;
4        public static final int EXTERN_DELAYED = 998;
5        public static final int EXTERN_CURRENT = 999;
```

Listing 6.5: Possible *input modes* of a *TaskQueueItem*

Building a *TaskQueueItem* without any configuration means to build a single independent unit. To be more specific, the attribute *NONE* is defined as the default *input mode* of each

TaskQueueItem as it is not always necessary that a task requires any input. However, it can be extremely helpful to create dependencies between single tasks or to pass data to an *SyncTask* before it is executed. A dependency between two tasks means that the output of a previous task is passed as the input to the following task. This can be realized by using the functions *setInput(TaskQueueItem item)* or *addInput(TaskQueueItem item)*. The necessity for both methods is that a *TaskQueueItem* can contain a single input as well as multiples ones. However, it is ensured that a task receives the result of a previous item as all tasks are executed synchronously before. Besides the dependency between *TaskQueueItems*, an item can also expect an external input that is not currently available. Therefore the flag *EXTERN_DELAYED* can be used. Whenever the *input mode* of a *TaskQueueItem* is set to this flag, it will wait until the input is received. This concept could be useful if the overall procedure needs information from a user. The last flag shown in Listing 6.5, is set whenever information can be passed directly to the *TaskQueueItem*. An example is shown in the following Listing 6.6.

```
1  new TaskQueueItem.Builder()
2       .setTask(new ListIntentTask())
3       .setInput(new IntentData(workspaceId))
4       .setOutputDataConverter(new TaskQueueItem.
           OutputDataConverter<IntentCollection,List<String>>() {
5
6          @Override
7          public List<String> parseData(IntentCollection input) {
8                  List<String> result = new ArrayList<>();
9                  for (IntentExport i : input.getIntents())
10                     result.add(i.getIntentName());
11                 return result;
12             }
13         })
14      .build();
```

Listing 6.6: Sample of creating a *TaskQueueItem*

To create a *TaskQueueItem* it is necessary to set a *SyncTask* or otherwise the task would not have any functionality. In the sample above, the task lists all existing Intents of *Workspace* with the given workspace id. The use of the *setInput(D data)* method sets the input mode automatically to *EXTERN_CURRENT*. Therefore, the *TaskQueueItem* knows, that it can directly access the passed data and does not have to wait for it. However, the *TaskQueueItem* will transmit its results to its successors as soon as it was executed successfully. This leads to the conclusion that the result must be suitable for the following task. Therefore the data structure must be adapted. This can be realized by the use of the *OutputDataConverter*. The *interface* allows the conversion of an incoming data element into a useful structure. As the sample above illustrates, the *TaskQueueItem* expects an *IntentCollection* and transforms it into a list that contains all names of the found *Intent*. This list is used to display the already existing *Intent*s to the expert. Afterwards, the second task is waiting for the *Intent* name which is entered by the expert.

Due to the fact that procedures could include decisions, it is possible to add an *ExecutionCondition*. As the name suggests, it can be used to define whether the task should be executed or not. In the procedure of Figure 5.10, it is used to decide whether a new *Intent* has to be created or if it must be only extended by an *Example*. The condition can also be implemented during the definition of a *TaskQueueItem*, as it is shown in Listing 6.6.

After all tasks of a process are defined and registered to a *TaskQueue*, they can be executed gradually by calling the function *executeTaskQueue* of the *ConversationManager*. This is shown in Listing 6.7. The *TaskQueueCallback* that is passed to the mentioned function can be used to perform actions during the process. Therefore each *TaskQueueItem*, as well as its index, can be accessed before and after its execution. This could be used to pass data to the tasks or react to the intermediate results. During the process of extending the knowledge base, the *onPreExecution* and the *onPostExectution* functions are used to display introductions to the user. Depending on the result of a single *TaskQueueItem*, the displayed text must be assembled first or it could be displayed without any modification.

The result of the overall procedure is returned via the *onResult* method. The type of the result can be defined by declaring the bounded type parameter of the *TaskqueueCallback*. This function is only triggered if the process is completed successfully. Afterwards the *TaskQueue* is finished but can be restarted if necessary.

```
1   conversationManager.executeTaskQueue
2               (taskQueue, new TaskQueueCallback<String>() {
3
4                   @Override
5                   public void onPreExecution(TaskQueueItem
                        queueItem, int index) {
6                       //handle queueItem before execuction
7                   }
8
9                   @Override
10                  public void onPostExecution(TaskQueueItem
                        queueItem, int index) {
11                      //handle queueItem after execuction
12                  }
13
14                  @Override
15                  public void onResult(String result) {
16                      //handle result
17                  }
18              });
```

Listing 6.7: Code snippet to demonstrate the *TaskQueueCallback* and the *executeTaskQueue*-function

6.3 ConversationCase

The broad process regarding to *ConversationCases* is already explained in 5.2.3. To summarize it, different types of cases can be identified and handled individually as soon as a situation arises. However, the system can only determine the current existing case if it is predefined. This section demonstrates how cases can be defined and managed, so they can be matched with the occurring situation.

The general approach of managing a case requires the definition of a concrete situation. This indicates that before the system can figure out if a situation occurred, it must be defined first. The sample in Listing 6.8 defines a *SuicideCase*. That occurs if a user enters words or phrases which can be associated with suicide.

```
{
  "id" : "a56381fc-93e1-11e8-9eb6-529269fb1459",
  "title" : "Warning",
  "description" : "Possible escalation detected",
  "type" : 0,
  "keyWords" : ["Suicide"]
}
```

Listing 6.8: Definition of the *SuicideCase* that can be requested from the server

As such a case is handled by sending a *Push Notification* to the experts, its data structure contains a *title* and *description* that is shown to the expert as soon as the notification is received. Whereas the mentioned attributes are only required to improve the usability of the system, the *keyWords*-attribute is necessary to identify the case as a *SuicideCase*.

```
1    public int compareTo(@NonNull MessageResponse response) {
2        List<RuntimeEntity> entities = response.getEntities();
3        for (RuntimeEntity entity : entities) {
4            for (String keyWord : getKeyWords()) {
5              if (entity.getEntity().equals(keyWord)) return 0;
6            }
7        }
8        return -1;
9    }
```

Listing 6.9: Function to decide if it the case occured

However, the defined keywords are not referring to a word that is included in an entered phrase, but rather refer to an *Entity*. This is shown by the the code snippet in Listing 6.9.

As soon as the mobile application receives a response from *IBM Watson Assistant*, it needs to get the associated *ConversationCase*. Therefore the function *getCase(T compareObject)* of the class *ConversationCaseManager* can be used. This method loops over all registered cases and calls its *compareTo* function. As it can be seen in Listing 6.9, the function of this specific case expects a *MessageResponse* from the chatbot, gets the detected *Entities* and compares it with the keywords. If one *Entity* name matches with the keyword "Suicide", the current situation is identified as a *SuicideCase*. Afterwards the *handleCase(I input)* function is called.

```
1    @Override
2    public void handleCase(MessageResponse input) {
3        final CaseInstance caseInst = createCaseInstance();
4
5        FirestoreCaseHandler.getInstance().createCaseInstance(
            caseInstance, new OnSuccessListener<Void>() {
6            @Override
7            public void onSuccess(Void aVoid) {
```

```
8                   sendPushNotification(caseInstance);
9               }
10          }, new OnFailureListener() {
11              @Override
12              public void onFailure(@NonNull Exception e) {
13                  Log.e(TAG, e.getMessage());
14              }
15          });
16      }
```

Listing 6.10: Function *handleCase* of class *PushCase*

The code snippet in Listing 6.10 represents the *handleCase* function of the class *PushCase*. The name suggests that the case is handled by sending a *Push Notification*. However, before that can be done, it is necessary to create a *CaseInstance*. Its structure is illustrated by Listing 6.11, as it has not yet been presented in Section 5.2.3.

```
1 {
2   "id" : "a56381fc-93e1-11e8-9eb6-529269fb1459",
3   "caseId" : "d23eb99d-d689-4653-b15f-2015579235aa",
4   "topicId" : "4d918109-66da-4700-b6f6-a161c95dcf2e",
5   "chatHistoryId" : "d23eb99d-d689-4653-b15f-2015579235aa",
6   "participant" : {...}
7 }
```

Listing 6.11: Structure of a *CaseInstance*

The *CaseInstance* contains all necessary data that is required to associate the occurred case with a topic, the conversation itself and its participants. Each *CaseInstance* is stored in a database. Therefore *Cloud Firestore* is used. Besides the CaseInstance, the *NoSQL cloud database* is used to store the predefined *ConversationCases*, *ChatHistory*, *Topics* and *Users*.

Afterwards a *Push Notification* is sent to the experts containing the identifier of the *CaseInstance*. As soon as the expert touches the notification, the mobile application on the side of the expert requests the *CaseInstance* to get all necessary data about the case and to handle it.

To keep the overall system flexible, all mentioned system components were designed as generic as possible. This is necessary as the technology could evolve and change in the future. This affects the already used service during this master's thesis as well as the temporary inspected services, that are provided by enterprises like *Google* or *IBM*. However, the use of this services can help to improve the interaction between systems and human beings. Due to time constraints, it was not possible to put these approaches into practice. The possible improvements of a system which are described in this thesis, are explained in the next chapter.

7

Future Prospect

To improve the mentioned system could mean to increase its usability, extend its knowledge base or develop it to become more human-like. As the general approach of this thesis has the aim of implementing a virtual assistant that can cover different scenarios in the medical or psychological areas but does not want to replace specialists, this chapter discusses the first two of the listed aspects.

7.1 Universal Chatbot

The mobile chatbot application that was developed during this master's thesis can connect to multiple third-party services. As each chatbot has its own knowledge base, they are handled differently. Referring to *IBM Watson Assistant* each topic is represented by an own *Skill*. However, creating a universal chatbot that is associated with multiple *Skills* could be a new approach to improve the usability of the mobile application. That means that one chatbot contains knowledge about all topics. Special cases, such as differentiate phrases which relate to different topics, can complicate the creation of an accurate system. A second approach would be to choose topics via chat. Instead of picking a topic from a list, a guided dialog between a user and the system could help to choose the topic. This conversation could be performed by one chatbot that contains information about all existing subjects. After a topic is chosen by the user, the system connects with the associated workspace.

7.2 Make it smarter

The generalized title of this section relates to the extension of knowledge. This does not only include extending the knowledge base as it is defined in Section 5.3, but also using other information from medical or psychological areas, for example the usage of emotional input. Another approach is to simplify the process of how knowledge is added to the knowledge base. Therefore *IBM* and *Google* provide different functionality. Due to time constraints, it was not possible to inspect all possible third-party services that can be used to improve the overall system. Therefore, the amount of possible extra features is reduced to *IBM Watson Tone Analyzer*, extending the knowledge base by using *IBM Watson Natural Language Understanding* and creating user profiles with *IBM Watson Personal Insights*.

7.2.1 IBM Watson Tone Analyzer

IBM Watson Tone Analyzer can be used to detect emotional and language tone in text. As *IBM* says, the service can be used to understand how a written communication is perceived [40]. In the context of this master's thesis it could be used to analyze written input from the user to create information about his or her current feelings. In the psychological sector, such data could help to adapt the system responses depending on the users emotional state. As an example, a user enters the phrase "I'm fine, thanks" but the *IBM Watson Tone Analyzer* detected *Anger* in the previous conversation, so the system could response with "Are you sure?" instead of "Great!". However, the amount of different tones is restricted. The following sample demonstrates the different types and a possible result that is created by the *IBM Watson Tone Analyzer*.

"I'm glad that I can talk to you. I had a fight with my wife yesterday and i feel so sad right now."

As *IBM* provides a graphical user interface to test single services, the phrases above could be simply analyzed by *IBM Watson Tone Analyzer*. The service analyzes the following tones on Document-level and Sentence-level: *Anger*, *Fear*, *Joy*, *Analytical*,

Confident and *Tentative*. The validation of the sample phrase above leads to the detection of *Sadness* and *Tentative* which can be equalized with sad and doubtful [41].

As already mentioned, such data can be used to create user profiles or to specify the system responses. However, as tones can change per user and per conversation, the data should not be stored in the knowledge base.

7.2.2 Complement the knowledge base

In contrast to emotional information, knowledge about a specific topic should be stored in the knowledge base. However, the procedure about how data can be added to the knowledge base should be improved as the manual approach can be very time-consuming.

Enterprises like *IBM* and *Google* already provide beta or released services that could be used to extend the knowledge base. As *Google* introduces the *Knowledge Connector*, *IBM* offers the *Natural Language Understanding* that can be combined with the *Knowledge Studio*.

The beta version of *Knowledge Connector* allows to complement already defined *Intents*. Therefore, multiple knowledge bases representing collections of documents, can be created. Each document can be parsed to find responses. At the moment of this research, *Google* supports the analysis of FAQ and *Knowledge Base Articles* [42]. The figure below shows a result of the parsing process of the *Google Privacy & Terms FAQ*.

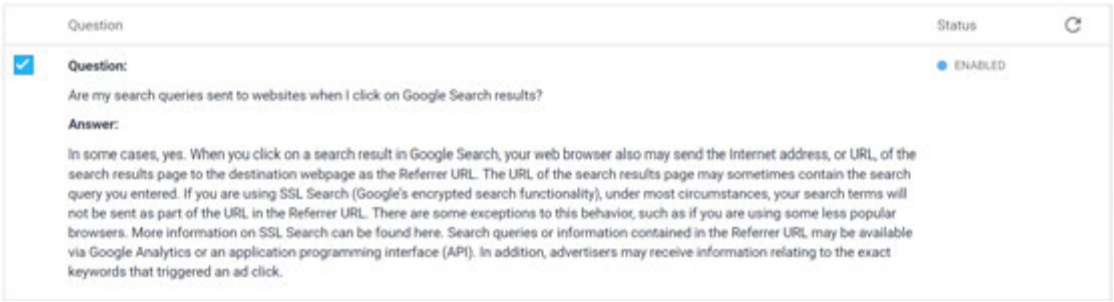


Figure 7.1: Question and Answer as a result of the parsing process of *Knowledge Connector*

Furthermore, the result can be converted into a defined *Intent*. That means that questions are added as *training phrases* and answers as *responses* [42].

As already mentioned, *IBM* provides services that can be used to complement the knowledge base as well. Using *IBM Watson Natural Language Understanding* allows to analyze semantic features of text input. This indicates that it can determine *categories*, *concepts*, *emotions*, *entities*, *keywords*, *metadata*, *relations*, *semantic roles*, and *sentiment* from plain text or webpages [43]. Similar to Section 7.2.2, *IBM* provides a demonstration of this service. Analyzing the *Wikipedia* entry about "test anxiety" results into the following output that is shown in figure 7.2.

Name	Type	Score
Cognitive Test Anxiety	PrintMedia	0.97
situational anxiety	HealthCondition	0.81
severe anxiety	HealthCondition	0.74
executive	JobTitle	0.49
George Mandler	Person	0.48
Irwin G. Sarason	Person	0.39

Figure 7.2: Found entities after analyzing *Wikipedia* entry about "test anxiety"

These results can be used to complement the knowledge base. As an example, different phrases that relate to *HealthCondition* could be added as *synonyms*. However, the result also depends on the associated machine learning model. *IBM Watson Knowledge Studio* can be used to create such models and integrate them to the *IBM Watson Natural Language Understanding* service [44]. Consequently, machine learning models can be created for custom topics and for analyzing documents to find important segments. This theoretical aspect leads to the approach to use a combination of both services to complement a knowledge base.

7.2.3 User Profiles

Besides helping the system by simplifying the complement of the knowledge base, collecting user data could also help the expert. The access to additional user data could be realized by using the *IBM Watson Personality Insights*. The service can determine insights about personality characteristics from social media, enterprise data, or other digital communications [45]. As an example, the mobile application could provide a functionality that allows the user to sign in via *Facebook*. Afterwards, his profile could be used to determine data that could be important for specific topics. The following Figure 7.3 illustrates what kind of data could be derived.

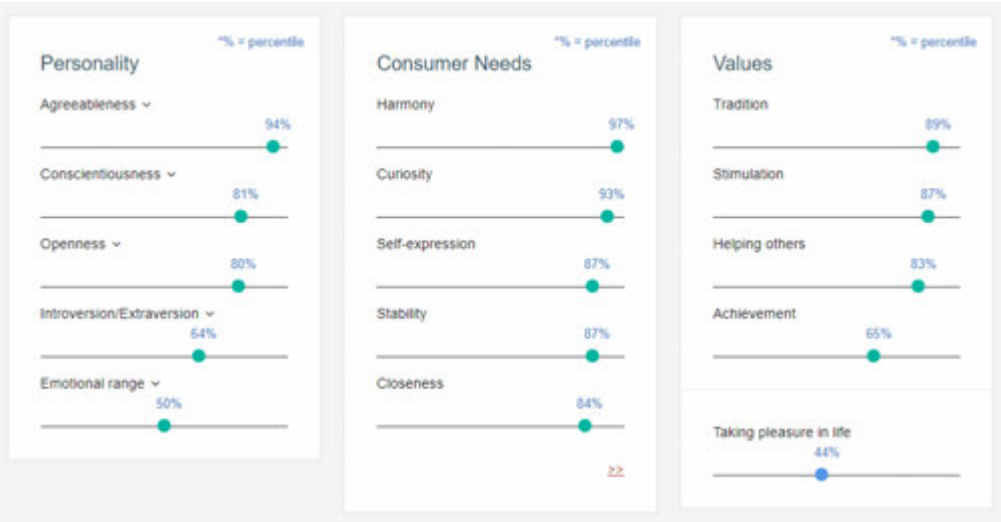


Figure 7.3: Personality characteristics after analyzing the *Twitter*-Account of *@Oprah (EN)*

8

Conclusion

The objective of this master's thesis was to create a generic system that supports patients and professionals in the medical or psychological sector. The overall system is split into three system components: mobile chatbot application, back-end application and central knowledge base. Due to time constraints, it was only possible to implement the mobile application that is used to test the developed frameworks which are explained in 5.2. However, theoretical concepts of the knowledge base and the back-end application are introduced and may be proven in further scientific work.

The back-end application and the central knowledge base became part of the overall system after it turned out that *IBM Watson* and its knowledge base that is trained for the game show *Jeopardy!* is not accessible. Instead, *IBM* provides individual services to build or to improve a chatbot. As each instance of a chatbot is connected with its own knowledge base but the separation of multiple knowledge bases leads to redundancy of data elements, the approach of a central knowledge base became relevant. Furthermore, one huge knowledge base can improve the chatbots' usability by avoiding the separation of multiple topics. However, each chatbot is only useful if it contains knowledge. The quality of the chatbot is directly reflected by the quality and quantity of the deposited data. In short, if the chatbot does not contain the knowledge about depression, a depressive patient who asks for help may find the system meaningless. As the knowledge base is an important aspect of the overall system, the back-end application is required to provide experts with the functionality to add knowledge. Furthermore, it can be used to manage user roles or as an interface that communicates with different chatbot services in the future.

Besides the theoretical concepts, the concept and the architecture of the mobile application were developed and implemented. It represents the virtual assistant that can be used by patients and experts. This application allows the user to interact with it by asking questions about a specific topic. Furthermore, the mobile application detects situations that depart from a default conversation. Within the scope of this master's thesis, phrases that relate to suicidal thoughts or unknown user input represent special situations and are used to test the implemented concept. To improve the readers understanding about the precise result of this master's thesis, the figures below demonstrate a possible conversation between a user and the chatbot. However, the content of the dialog was created by *Eileen Bendig*, a doctoral student at the Department of Clinical Psychology and Psychotherapy, Ulm University. The dialog was written in German and it is based on the literature *ACT-Training: Acceptance & Commitment Therapie of Jason Luoma, Steven C. Hayes and Robin D. Walser* [46].

All in all it can be said that the future-oriented topic of this thesis builds an approach of improving the health care by using AI based algorithms and concepts of handling emergencies. However, there is still a lot to be done to make the system ready for use. Therefore, it needs to be further developed and the knowledge base needs to be extended by the support of medical and psychological professionals and one day it will hopefully support them, too.

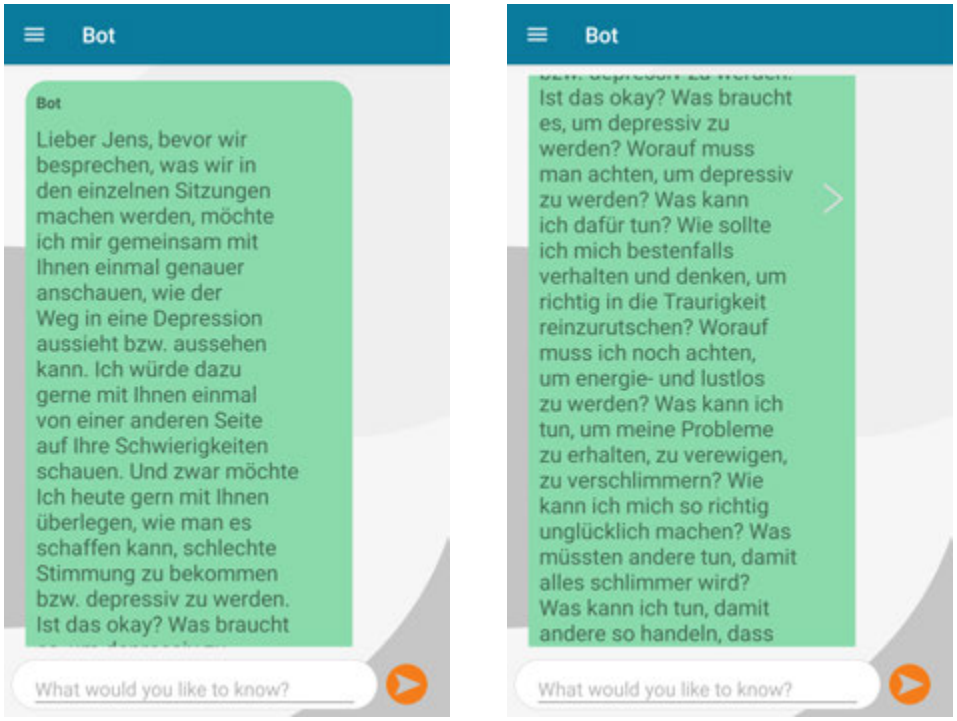


Figure 8.1: Possible conversation between patient and chatbot (A)

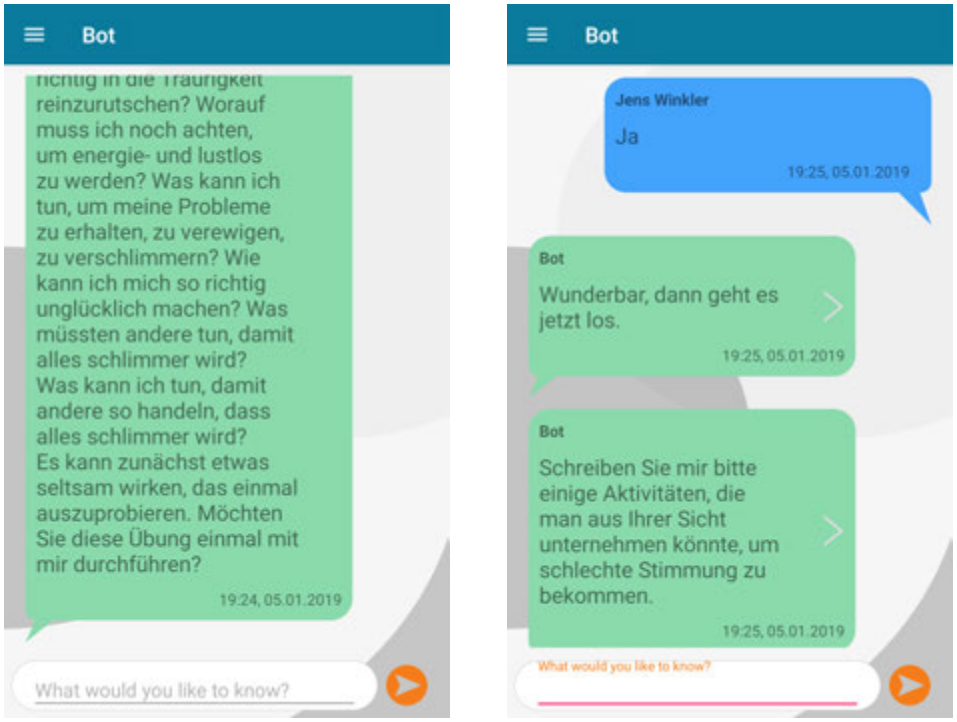


Figure 8.2: Possible conversation between patient and chatbot (B)

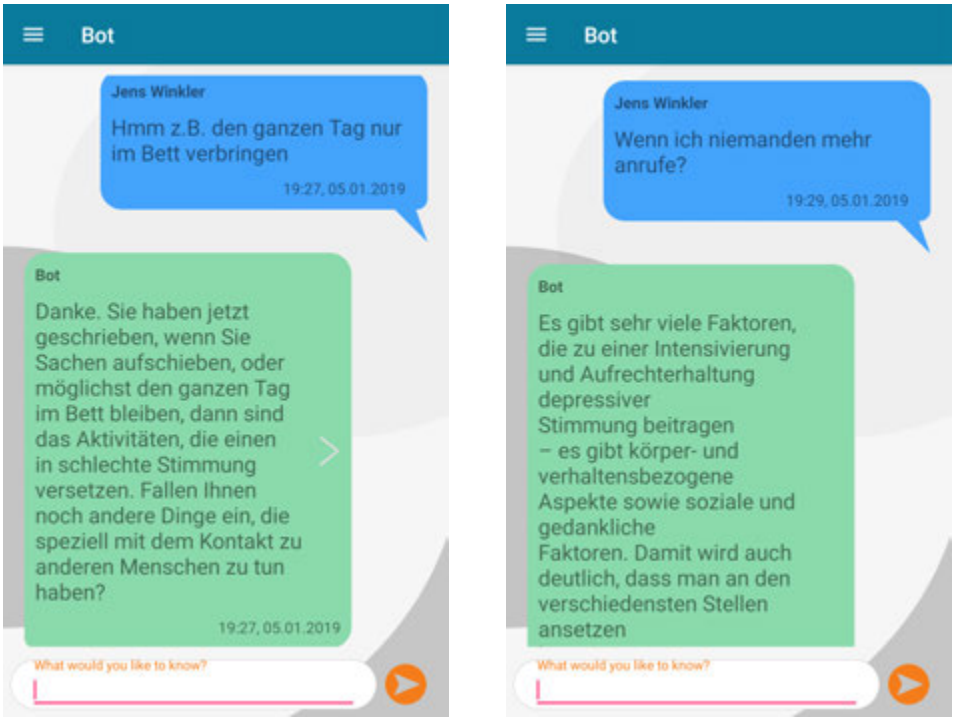


Figure 8.3: Possible conversation between patient and chatbot (C)

Figure 8.4: Possible conversation between patient and chatbot (D)

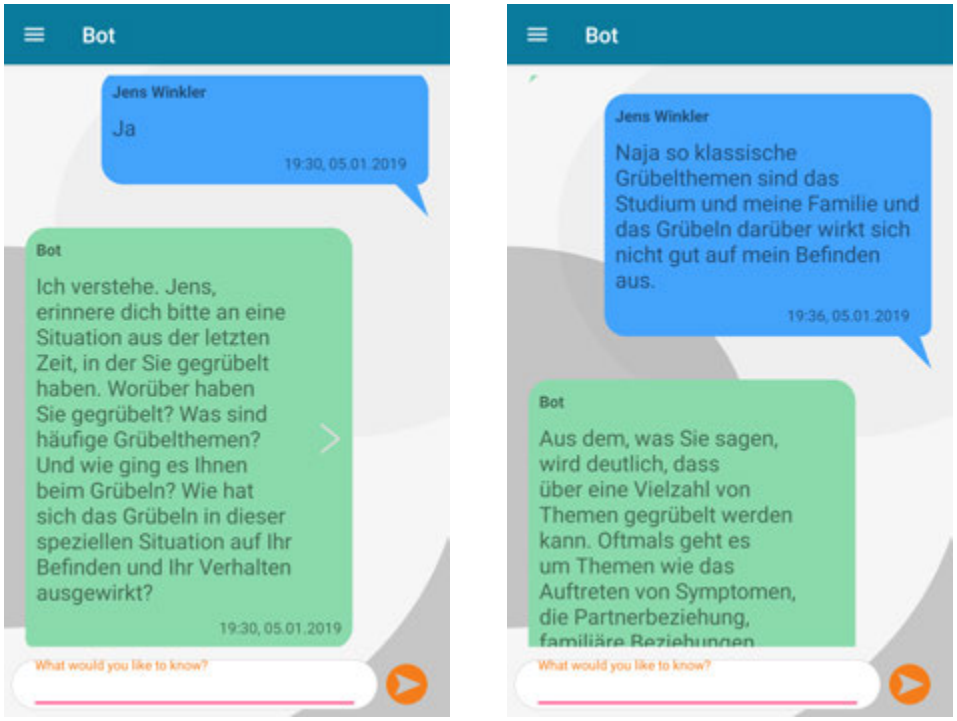


Figure 8.5: Possible conversation between patient and chatbot (E)

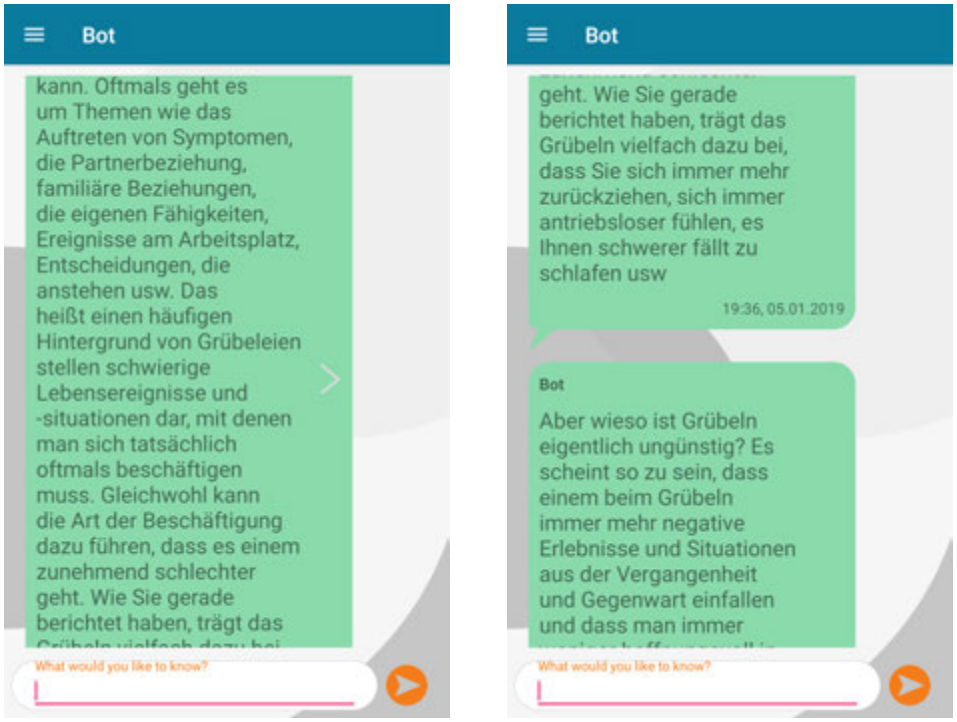


Figure 8.6: Possible conversation between patient and chatbot (F)

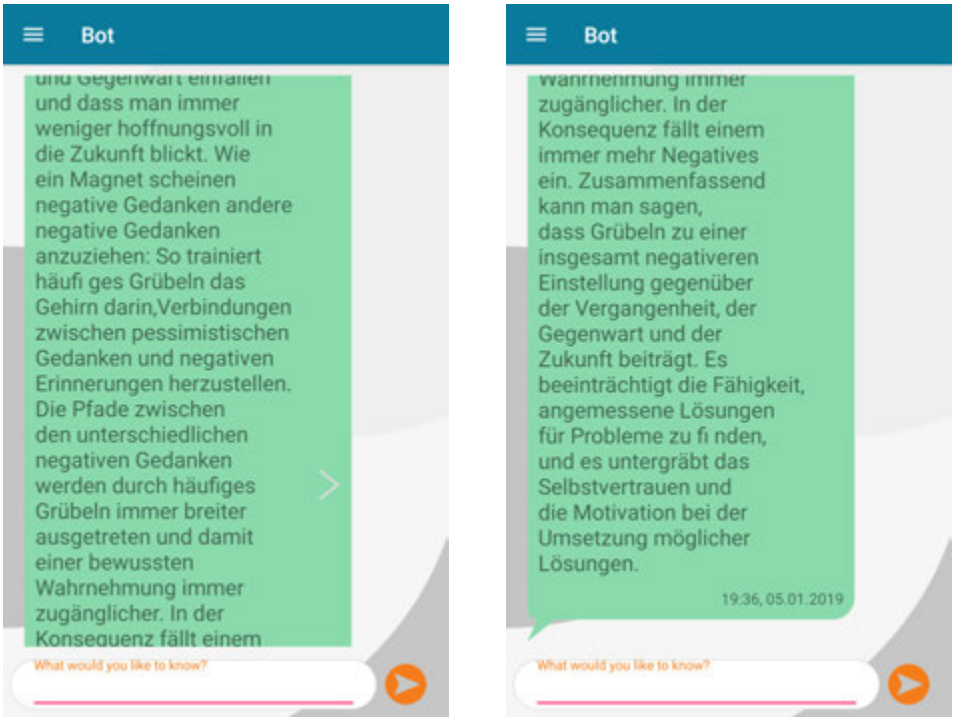


Figure 8.7: Possible conversation between patient and chatbot (G)

Bibliography

[1] *Chatbot Market Size To Reach $1.25 Billion By 2025*. August 2017. URL: https://www.grandviewresearch.com/press-release/global-chatbot-market (visited on 01/03/2019).

[2] *Global Views on Healthcare - 2018*. September 2018. URL: https://www.ipsos.com/sites/default/files/ct/news/documents/2018-07/global_views_on_healthcare_2018_-_graphic_report_0.pdf (visited on 01/03/2019).

[3] Ruth Robertson, John Appleby, and Harry Evans. *Public satisfaction with the NHS and social care in 2017*. February 2018. URL: https://www.kingsfund.org.uk/publications/public-satisfaction-nhs-2017 (visited on 01/03/2019).

[4] *Healthcare-Barometer*. March 2018. URL: https://www.pwc.de/de/gesundheitswesen-und-pharma/pwc-ergebnisse-healthcare-barometer-2018-final.pdf (visited on 01/03/2019).

[5] *The 2018 State of Chatbots Report*. January 2018. URL: https://www.drift.com/wp-content/uploads/2018/01/2018-state-of-chatbots-report.pdf (visited on 01/03/2019).

[6] *What Consumers Really Think About AI: A Global Study*. April 2017. URL: https://www.pega.com/sites/pega.com/files/docs/2017/Apr/what-consumers-really-think-about-ai.pdf (visited on 01/03/2019).

[7] Prof. Dr. Oliver Bendel. *Revision von Chatbot*. 2018. URL: https://wirtschaftslexikon.gabler.de/definition/chatbot-54248/version-277297 (visited on 12/05/2018).

[8] Rashid Khan and Anik Das. *Build better chatbots: A complete guide to getting started with chatbots*. New York, New York: Apress, 2018. ISBN: 978-1-4842-3111-1. URL: http://proquest.tech.safaribooksonline.de/9781484231111.

[9] Crina Grosan and Ajith Abraham. *Intelligent Systems: A Modern Approach*. Vol. 17. Intelligent Systems Reference Library. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2011. ISBN: 9783642210037. DOI: 10.1007/978-3-642-21004-4. URL: http://www.eblib.com/patron/FullRecord.aspx?p=769965.

[10] Bayan AbuShawar and Eric Atwell. "ALICE Chatbot: Trials and Outputs". In: *Computación y Sistemas* 19.4 (2015). ISSN: 1405-5546. DOI: 10.13053/cys-19-4-2326.

[11] Richárd Krisztián Csáky. *Deep Learning Based Chatbot Models*. 2017. DOI: 10.13140/RG.2.2.21857.40801.

[12] Joseph Weizenbaum. "Computational Linguistics: ELIZA - A Computer Program for the Stufy of Natural Language Communication Between Man And Machine". In: (1966). URL: https://cse.buffalo.edu/~rapaport/572/S02/weizenbaum.eliza.1966.pdf (visited on 12/05/2018).

[13] Maria João Pereira et al. *Chatbots' Greetings to Human-Computer Communication*. 2016. URL: http://arxiv.org/pdf/1609.06479v1.

[14] Mohammad Majid Al-Rifaie. *AISB - The Society for the Study of Artificial Intelligence and Simulation of Behaviour - Loebner Prize*. n.y. URL: https://www.aisb.org.uk/events/loebner-prize (visited on 12/09/2018).

[15] Jake Frankenfield. *Turing Test*. n.y. URL: https://www.investopedia.com/terms/t/turing-test.asp (visited on 12/09/2018).

[16] Apple. n.y. URL: https://www.apple.com/siri/ (visited on 12/07/2018).

[17] Jon Walker. *Chatbot Comparison – Facebook, Microsoft, Amazon, and Google - Artificial Intelligence Companies, Insights, Research*. 2017. URL: https://emerj.com/ai-sector-overviews/chatbot-comparison-facebook-microsoft-amazon-google/ (visited on 12/07/2018).

[18] KHARI JOHNSON. *Facebook promises to 'massively' simplify Messenger in 2018*. January 2018. URL: https://venturebeat.com/2018/01/16/facebook-promises-to-massively-simplify-messenger-in-2018/ (visited on 12/10/2018).

[19] Brian Rathjen and Jwelch742. *Jeopardy! (TV Series 1984–) - Plot Summary - IMDb*. n.y. URL: https://www.imdb.com/title/tt0159881/plotsummary (visited on 12/11/2018).

[20] *IBM100 - A Computer Called Watson: Technical Breakthroughs*. n.y. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/watson/breakthroughs/ (visited on 12/11/2018).

[21] *IBM100 - A Computer Called Watson: Overview*. n.y. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/watson/ (visited on 12/07/2018).

[22] *The DeepQA Research Team - IBM*. n.y. URL: https://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2159 (visited on 12/11/2018).

[23] Chris Higgins. *"What is IBM Watson?" 7 Videos from the Jeopardy! Era*. n.y. URL: http://mentalfloss.com/article/51543/what-ibm-watson-7-videos-jeopardy-era (visited on 12/11/2018).

[24] *IBM100 - A Computer Called Watson: Transforming the World*. n.y. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/watson/transform/ (visited on 01/13/2019).

[25] IBM. *About*. n.y. URL: https://console.bluemix.net/docs/services/assistant/index.html?locale=en#about (visited on 12/12/2018).

[26] IBM. *Informationen*. n.y. URL: https://console.bluemix.net/docs/services/assistant/index.html#informationen (visited on 12/12/2018).

[27] IBM. *Release notes*. n.y. URL: https://console.bluemix.net/docs/services/assistant/release-notes.html?locale=en#releaseinformationen (visited on 12/12/2018).

[28] IBM. *Skills*. n.y. URL: https://console.bluemix.net/docs/services/assistant/skills.html#skills (visited on 12/12/2018).

[29] IBM. *Building a client application*. n.y. URL: https://console.bluemix.net/docs/services/conversation/develop-app.html (visited on 12/13/2018).

[30] Chin-Yuan Huang et al. “A Chatbot-supported Smart Wireless Interactive Healthcare System for Weight Control and Health Promotion”. In: *2018 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. IEEE, 12/16/2018 - 12/19/2018, pp. 1791–1795. ISBN: 978-1-5386-6786-6. DOI: 10.1109/IEEM.2018.8607399.

[31] Akihiro Yorita et al. “A Robot Assisted Stress Management Framework: Using Conversation to Measure Occupational Stress”. In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 10/7/2018 - 10/10/2018, pp. 3761–3767. ISBN: 978-1-5386-6650-0. DOI: 10.1109/SMC.2018.00637.

[32] Haolin Wang et al. “Social Media–based Conversational Agents for Health Management and Interventions”. In: *Computer* 51.8 (2018), pp. 26–33. ISSN: 0018-9162. DOI: 10.1109/MC.2018.3191249.

[33] Wei-De Liu, Kai-Yuan Chuang, and Kuo-Yi Chen. “The Design and Implementation of a Chatbot’s Character for Elderly Care”. In: *2018 International Conference on System Science and Engineering (ICSSE)*. IEEE, 6/28/2018 - 6/30/2018, pp. 1–5. ISBN: 978-1-5386-6285-4. DOI: 10.1109/ICSSE.2018.8520008.

[34] *The Top 12 Health Chatbots - The Medical Futurist*. 2018. URL: https://medicalfuturist.com/top-12-health-chatbots (visited on 01/05/2019).

[35] *Woebot - Your charming robot friend who is here for you, 24/7*. n.y. URL: https://woebot.io/the-science (visited on 12/30/2018).

[36] *Woebot - Your charming robot friend who is here for you, 24/7*. n.y. URL: https://woebot.io/faqs/ (visited on 12/05/2018).

[37] Kathleen Kara Fitzpatrick, Alison Darcy, and Molly Vierhile. “Delivering Cognitive Behavior Therapy to Young Adults With Symptoms of Depression and Anxiety Using a Fully Automated Conversational Agent (Woebot): A Randomized Controlled Trial”. In: *JMIR mental health* 4.2 (2017), e19. ISSN: 2368-7959. DOI: 10.2196/mental.7785.

[38] Salman Razzaki et al. *A comparative study of artificial intelligence and human doctors for the purpose of triage and diagnosis*. 27.06.2018. URL: http://arxiv.org/pdf/1806.10698v1 (visited on 01/02/2019).

[39] *| Babylon Health*. n.y. URL: https://www.babylonhealth.com/product/healthcheck (visited on 01/02/2019).

[40] IBM. *About*. n.y. URL: https://console.bluemix.net/docs/services/tone-analyzer/index.html#about (visited on 12/28/2018).

[41] IBM. *IBM Watson Tone Analyzer*. n.y. URL: https://tone-analyzer-demo.ng.bluemix.net/ (visited on 01/02/2019).

[42] *Knowledge Connectors | Dialogflow Enterprise Edition Documentation | Google Cloud*. n.y. URL: https://cloud.google.com/dialogflow-enterprise/docs/knowledge-connectors (visited on 12/29/2018).

[43] *About*. n.y. URL: https://console.bluemix.net/docs/services/natural-language-understanding/index.html#about (visited on 12/29/2018).

[44] *About*. n.y. URL: https://console.bluemix.net/docs/services/watson-knowledge-studio/index.html#wks_overview_full (visited on 12/29/2018).

[45] *Getting started tutorial*. n.y. URL: https://console.bluemix.net/docs/services/personality-insights/getting-started.html#gettingStarted (visited on 12/29/2018).

[46] Jason Luoma et al. *ACT-Training: Handbuch der Acceptance & Commitment Therapie ; ein Lernprogramm in 10 Schritten*. Reihe Fachbuch ACT für die klinische Praxis. Paderborn: Junfermann, 2009. ISBN: 9783873877009. URL: http://deposit.d-nb.de/cgi-bin/dokserv?id=3090530&prov=M&dok_var=1&dok_ext=htm.

A

Sources

```
1  class DeleteIntentTask implements AsyncTask<IbmWatsonModule,
           ConversationData<String>> {
2
3      @Override
4      public void execute(final IbmWatsonModule module,
           ConversationData<String> data, final
5      TaskCallback callback) {
6
7          DeleteIntentOptions options = new DeleteIntentOptions.
               Builder(data.get(IntentData                .
               KEY_WORKSPACE_ID),
8                    data.get(IntentData.KEY_NAME)).build();
9
10         module.getUtil().deleteIntent(options).enqueue(new
               ServiceCallback<Void>() {
11              @Override
12              public void onResponse(Void response) {
13                  callback.onCallback(response);
14              }
15
16              @Override
17              public void onFailure(Exception e) {
```

```
18                      module.getErrorListener().onError(e.getMessage
                            ());
19                  }
20              });
21      }
22  }
```

Listing A.1: *DeleteIntentTask* as part of *IbmWatsonModule* configuration

List of Figures

List of Tables

Name: Jens Winkler Matriculation number: 937219

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, ...

Jens Winkler