DAT410 - Module 7: Dialogue Systems - Group 49

Harry Denell
MPDSC
[redacted]

Elvina Fahlgren
MPDSC
[redacted]

March 2025

"We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions."

1 Reading and summary

GUS was a early experimental dialog system, which was frame driven and designed to handle natural language conversations within an narrowly defined context ([1]). Specifically, it was used as planning simple round trip travel in California.

The motivation behind creating GUS was not to solve the practical task of travel planning itself but to investigate challenges in natural dialogue systems. The primary goal was to manage unpredictability in conversation. GUS tackled this through modularity, breaking down complex language understanding tasks into smaller components. Each component addressed specific linguistic and interactive issues such as interpreting user intentions, resolving ambiguous references (anaphora), handling indirect responses, and reconstructing incomplete sentence fragments ([1]).

An important aspect of GUS was mixed initiative conversation, allowing both the system and the user to alternate control of the interaction. Unlike earlier dialogue systems, which tended to dominated conversations, GUS provided more flexibility for natural conversational shifts, which enhanced interaction realism compared to with another human. Indirect user responses were also addressed, although within some limited scenarios. The system could interpret incomplete information through some well known conversational patterns and context. Anaphora in conversations was handled by GUS by relating ambiguous phrases or pronouns to previously mentioned entities or events. Additionally, GUS managed sentence fragments by using predefined sentence structures (skeletons) linked systematically to conversational context, ensuring incomplete user inputs are understood correctly ([1]).

Internally, GUS used structured data representations called frames. Frames organize information into slots, which store specific data or executable procedures. This modularity simplifies both debugging and knowledge management, and should make the system more extensible, without significantly more work. Attached procedures (which are essentially mappings of functions to slots), were divided into ‘demons’ (automatically triggered) and ’servants’ (activated upon request). These help with dynamic data handling, flexibly defining what should happen next, for a given slot and what the user has done ([1]).

A central ‘agenda’ managed tasks and priorities, allowing the system to flexibly return to earlier conversational stages if new information emerges. Although not highly intelligent, GUS highlighted key challenges and solutions for scalable language understanding. Future systems must balance guidance with flexibility, integrating large datasets, efficient retrieval, and structured knowledge for more advanced dialogue models ([1]).

2 Implementation

We implemented a simple text based digital assistant, which we called 'Ola Larsson' for a personal touch, that handles three tasks:

- weather forecasts
- finding restaurants
- providing transit information

To enable natural conversation rather than strict, menu-driven interactions, we chose a "frame-based" architecture inspired by GUS, by using keyword matching for intent detection. Each task is represented as a frame whose configuration is stored in a dictionary. For example, the weather frame requires the "location" and "date" slots, as shown in the following code snippet:

```
FRAME_CONFIG = {
    "weather": ("Weather", ["location", "date"]),
    "restaurant": ("Restaurant", ["location", "cuisine"]),
    "transit": ("Transit", ["location", "time"]),
}

def create_frame(frame_type):
    config = FRAME_CONFIG[frame_type]
    return Frame(config[0], config[1])
```

The system identifies user intent by matching input phrases against predefined keywords for each task:

```
KEYWORDS = {
    "weather": ["weather", "forecast", "temperature"],
    "restaurant": ["restaurant", "food", "eat", "dinner", "lunch", "cuisine"],
    "transit": ["bus", "tram", "transit", "train", "metro"]
}

def select_frame(user_input):
    text = user_input.lower()
    for key, keywords in KEYWORDS.items():
        if any(keyword in text for keyword in keywords):
            return key
    return None
```

Once identified, it instantiates a structured frame containing specific slots (e.g., location, date, cuisine) that must be filled to complete the user's request. The system uses a simple, but yet rather flexible extraction mechanism, which is based on keyword validation against predefined valid values loaded from external JSON files (see Appendix A.1), mimicking backend databases. These sample valid values were created with `GitHub Copilot` extension in VSCode. Code belows show this:

```
def update_frame(user_input, current_frame):
    if current_frame.name == "Weather":
        if current_frame.slots["location"] is None:
            candidate = extract_first_valid_value(user_input, "location")
            if candidate:
                current_frame.fill_slot("location", candidate)
        if current_frame.slots["date"] is None:
            candidate = extract_first_valid_value(user_input, "date")
            if candidate:
                current_frame.fill_slot("date", candidate)
    # similar for restaurant and transit
```

To handle partial user responses, the assistant employs a dynamic prompting mechanism. If any slot information is missing, it asks the user some set targeted questions, and providing random example values to guide valid responses. User inputs are validated, and if invalid, the system continues prompting until acceptable values are provided. This enables a more flexible and conversational interaction, allowing users to provide details incrementally or indirectly. The code for this is as follows:

```
def fill_missing_slots(current_frame, valid_data, validators):
    first_time = True
    while not current_frame.is_complete():
        question, slot = ask_for_missing_info(current_frame)
        if question:
            if first_time:
                print(f"{chat_bot_name}: {question}")
            user_response = input("You: ")
            # generic extraction on the user's response
            candidate = extract_first_valid_value(user_response, slot)
            if candidate is None or not validators[slot][0](candidate):
                examples = validators[slot][1]()
                print(f"{chat_bot_name}: I didn't recognize that {slot}. Please
                    provide a valid {slot} (e.g., {examples}).")
                first_time = False
                continue
            # use the extracted candidate for filling the slot
            current_frame.fill_slot(slot, candidate)
            current_frame = update_frame(user_response, current_frame)
            first_time = True
    return current_frame
```

After all required slots in the frame are filled, the system provides the final response (e.g., “Fetching weather for berlin on monday”) and then asks if the user needs further assistance.

The whole code is provided in Appendix A. Here is an example conversation with the chat bot:

```
Ola Larsson: Welcome to your digital assistant Ola Larsson. How can I help
you?
You: I want to eat
Ola Larsson: What location? (e.g., Berlin, New Orleans)
You: In Miami
Ola Larsson: What cuisine? (e.g., low-sodium, nigerian)
You: I want italian food
Ola Larsson: Searching for italian restaurants near miami.
Ola Larsson: Is there anything else I can help you with? (yes/no)
You: Yes
Ola Larsson: How can I help you?
You: I want to know the weather in Stockholm on Monday
Ola Larsson: Fetching weather for stockholm on monday.
Ola Larsson: Is there anything else I can help you with? (yes/no)
You: Yes
Ola Larsson: How can I help you?
You: I want to take the bus
Ola Larsson: What location? (e.g., Santiago, Stockholm)
You: In copenhagen
Ola Larsson: What time? (e.g., 5 am, 7 pm)
```

```
You: 3 am
Ola Larsson: Retrieving transit info for copenhagen at 3 am.
Ola Larsson: Is there anything else I can help you with? (yes/no)
You: no
Ola Larsson: Goodbye!
```

2.1 Weaknesses

There are several weaknesses of our initial system. Firstly, it relies on basic keyword matching to determine the user's intent. While this approach works for clear and simple inputs, it fails to capture nuances of natural language. For example, synonyms or slightly different phrasing may not trigger the correct functionality, leading to misunderstandings.

This method also lacks robustness when users provide indirect or ambiguous statements. As a result, the system might ask repetitive or inappropriate questions, as seen in cases where it repeatedly asks for a valid location or cuisine when the user is unsure. In general, this also does not scale well. At the moment, one would have to add all the keywords that we would want the system to catch, and that obviously does not scale too well. Although the generic extraction function improves upon simple string splitting, it still depends on exact matches from predefined valid lists. This rigidity can result in failure to recognize valid inputs, as previously said, that are slightly misspelled or phrased differently.

Further, the current validation method may frustrate users if they repeatedly provide input that the system rejects, without any learning mechanism to adapt over time. An example is seen below:

```
Ola Larsson: Welcome to your digital assistant Ola Larsson.
How can I help you?
You: wether
Ola Larsson: I'm sorry, I couldn't determine your request.
Please mention weather, restaurant, or transit.
You: weather
Ola Larsson: What location? (e.g., Mexico City, Santiago)
You: göteborg
Ola Larsson: I didn't recognize that location.
Please provide a valid location (e.g., Reykjavik, Oslo).
You: gothenburg
Ola Larsson: What date? (e.g., friday, next year)
You: frday
Ola Larsson: I didn't recognize that date.
Please provide a valid date (e.g., next week, sunday).
You: friday
Ola Larsson: Fetching weather for gothenburg on friday.
```

Also, the system wouldn't be able to give several solutions; for example, if the user were to ask for both Italian and Greek cuisine, it would just search for the first cuisine the user mentioned, e.g the first matching word.

Moreover, it is limited when it comes to context awareness. The system doesn't retain context beyond the current task. Consequently, it cannot manage follow up queries or adapt to topic shifts within a conversation. A more advanced dialogue system would need to maintain a persistent state across multiple turns to support a more natural, mixed-initiative conversation.

The system uses static, fictitious data (from a JSON file) for its responses. This means that the output is not based on real-time information and is limited in scope. For a production-ready digital assistant, integration with live APIs (for weather data, restaurant listings, and transit schedules) would be necessary, which would introduce additional challenges like handling asynchronous data and network latency.

Lastly, the chat bot isn't able to switch conversation when it is still waiting to receive answers from the user in order to determine an answer. Here is an example:

```
Ola Larsson: Welcome to your digital assistant Ola Larsson. How can I help
you?
You: I want to eat
Ola Larsson: What location? (e.g., Budapest, Vienna)
You: I want to know the weather in London
Ola Larsson: What cuisine? (e.g., gluten-free, french)
You: I want to know the weather
Ola Larsson: I didn't recognize that cuisine. Please provide a valid cuisine
(e.g., brazilian, palestinian).
You: What is the weather in London?
Ola Larsson: I didn't recognize that cuisine. Please provide a valid cuisine
(e.g., raw, low-sugar).
```

2.2 Further implementation

Building upon our initial somewhat simpler keyword based implementation, we treid further developed the assistant by adopting a more full on frame based approach inspired by the GUS dialogue system. Our goal was to enhance flexibility and extensibility while maintaining main functionality of the inital approach.

The idea is that this implementation should build on the previous functionality, and therefore behave the same. In most sense, it does, with some minor differences, see comment of weakness at end of this section. Further, we also played around with some of the chatbots phrases, like its welcome phrase, and the length and verbosity of its responses (like its hints when given invalid input), to get a feeling for this. An example conversation using this version could look like:

```
Ola Larsson: Welcome! How can I assist you today?
Ola Larsson: Please provide topic (e.g., weather, transit)
You: weather
Ola Larsson: Please provide the following: location, date
You: barcelona
Ola Larsson: Please provide date (e.g., friday, saturday)
You: tomorrow
Ola Larsson: Fetch weather for Barcelona on Tomorrow? (yes/no) yes
Ola Larsson: Fetching weather for Barcelona on Tomorrow...
Ola Larsson: Would you like to continue? (yes/no) yes
Ola Larsson: How can I assist you next?
Ola Larsson: Please provide topic (e.g., weather, restaurant)
You: bus transit
Ola Larsson: Please provide the following: location, time
You: madrid 9 am
Ola Larsson: Find transit options for Madrid at 9 am? (yes/no) no
Ola Larsson: Would you like to continue? (yes/no) yes
Ola Larsson: How can I assist you next?
Ola Larsson: Please provide topic (e.g., restaurant, weather)
You: i want to eat
Ola Larsson: Please provide the following: location, cuisine
You: italian
Ola Larsson: Please provide location (e.g., stockholm, houston)
You: barcelona
Ola Larsson: Search Italian restaurants in Barcelona (yes/no) yes
Ola Larsson: Searching Italian restaurants in Barcelona...
Ola Larsson: Would you like to continue? (yes/no) no
Ola Larsson: Goodbye!
```

The core idea involves structuring the dialogue system around hierarchical frames, that themselves can be made up of other frames. Frames are implemented as classes with a common abstract base, enabling easier extension to additional topics, as well as the recursivity of building frames within frames.

The `Frame` class serves as the foundational base structure for all components in the system. Each frame can either then be a 'leaf frame', holding a specific value, or a non-leaf frame, containing slots that themselves are frames (that could be both leaf and non-leaf, etc.). This recursive structure allows interactions to be broken down into manageable parts, while putting all functionality related to a certain level of abstraction of information for itself.

Below is the code for the abstract Frame class:

```python
# define the frame structure for the chatbot
# idea is to have a tree-like structure of frames, where each frame can have slots
# and each slot is a frame itself
# either a leaf-frame with a value or a non-leaf-frame with slots
class Frame(ABC):
    def __init__(self, name, valid_values=None, extract=None, validate=None):
        self.name = name
        self.value = None
        self.valid_values = valid_values
        self.extract = extract
        self.validate = validate
        self.slots = {}
        self.servants = []

    # check if the frame is a leaf
    def is_leaf(self):
        return not self.slots

    # check if the frame is complete
    # a frame is complete if it is a leaf and has a value
    def is_complete(self):
        if self.is_leaf():
            return self.value is not None
        else:
            return all(slot.is_complete() for slot in self.slots.values())

    # recursive function to fill frame and its slots with value extracted from the
        text
    # if the frame is a leaf, it tries to extract a value from the text and
        validate it against the valid values
    # if the frame is not a leaf, it tries to fill its slots
    def fill(self, text):
        if self.is_leaf():
            if self.value is None:
                candidate = self.extract(text, self.valid_values)
                if candidate and self.validate(candidate, self.valid_values):
                    self.value = candidate
                    # if the frame is a leaf and we have a value, we are done here
                    return True
            return False
        else:
            for slot in self.slots.values():
                slot.fill(text)
            # if all slots are complete, we can try to run the frame
            return self.is_complete()
    # this is our 'demon' for a frame
    # if the frame is not complete, it will prompt the user to provide the missing
        values
    # if the frame is a leaf, it will prompt the user to provide the value
    # else it will prompt the user to provide values for the missing slots
    # meaning it tries to fill several slots at once, on the same level
    def prompt(self):
        if self.is_leaf():
            while self.value is None:
                examples = ', '.join(random.sample(self.valid_values, min(2,
                    len(self.valid_values))))
                user_input = input(f"{chat_bot_name}: Please provide {self.name}
                    (e.g., {examples}) \nYou: ")
                if not self.fill(user_input):
                    print(f"{chat_bot_name}: Invalid {self.name}. Valid options: {',
                        '.join(self.valid_values.take(5))}.")
        else:
            missing = [name for name, slot in self.slots.items() if not
                slot.is_complete()]
            if missing:
                prompt_msg = f"{chat_bot_name}: Please provide the following: " + ",
                    ".join(missing) +"\nYou: "
                user_input = input(prompt_msg)
                for name in missing:
                    self.slots[name].fill(user_input)
                for slot in self.slots.values():
                    if not slot.is_complete():
                        slot.run(user_input)

    # method for running the frame
    # it will try to fill the frame with the initial text
    # if the frame is not complete, it will prompt the user
    # if the frame is complete, it will run the attached servants
    def run(self, initial_text):
        self.fill(initial_text)
        if not self.is_complete():
            self.prompt()
        # run servants onc complete / slots filled
        for servant in self.servants:
            servant(self)
```

Demons are the procedures triggered whenever required slot values are incomplete. The `prompt` method, together with the fill method, in each frame acts as its demon, guiding the user to provide necessary details until process of building out the frame is complete. This approach should ideally allow for more natural and flexible conversations, as users can provide information incrementally. This method also allows for fillling several slots for a certain frame level, meaning it should feel a little more dynamic to the user, than having to go through each frame abstraction level, providing more and more information.

Servants are the actions or functions attached to a frame that are executed once the frame is complete. For example, the `RestaurantFrame` has a servant that finds restaurants once all required slots are filled. It does this by first asking the user if they want to fetch the information that the sytem has gathered, as a confirmation. See code below:

```
# frame for restaurant
# it has a location and cuisine slot
# it has a servant that will find restaurants, if the user asks for it
class RestaurantFrame(Frame):
    def __init__(self):
       super().__init__("restaurant")
       self.slots["location"] = LocationFrame()
       self.slots["cuisine"] = CuisineFrame()
       self.servants.append(self.restaurant_servant)

   def restaurant_servant(self, frame):
       loc = frame.slots["location"].value.capitalize()
       cuisine = frame.slots["cuisine"].value.capitalize()
       answer = input(f"{chat_bot_name}: Search {cuisine} restaurants in {loc}
            (yes/no) ")
       if answer.lower() == "yes":
           print(f"{chat_bot_name}: Searching {cuisine} restaurants in {loc}...")
```

From this we also see an example of how the `RestaurantFrame` is made up of several other frames in its slots, e.g a `LocationFrame` and a `CuisineFrame`.

The main loop of the chatbot is responsible for initiating the conversation and managing the flow between different frames. See code below:

```
def main():
    print(f"{chat_bot_name}: Welcome! How can I assist you today?")
    while True:
        topic_frame = TopicFrame()
        topic_frame.run("")
        frame = topic_frame.get_frame()
        frame.run("")
        cont = input(f"{chat_bot_name}: Would you like to continue? (yes/no)
             ").strip().lower()
        if cont == "no":
            print(f"{chat_bot_name}: Goodbye!")
            break
        else:
            if cont == "yes":
                print(f"{chat_bot_name}: How can I assist you next?")
```

It starts by running the `TopicFrame`, which determines the user's intent and selects the appropriate task specific frame. The selected frame then takes over, prompting the user for any missing information and executing its servants once topic frame is complete. In general the design allows the system to rather flexibly handle inputs of varying completeness. Users can provide information incrementally or entirely within a single sentence, and the system dynamically adapts the questioning based on which slots remain unfilled.

Notably though, the user can with this implementation, only provide information for a certain 'level' of a frame. For instance beginning a discussion with 'weather barcelona tomorrow', will only catch 'weather' first. Then for the next prompt, the user can then enter 'barcelona tomorrow' and the chatbot will catch both of these, as they are part of the same slot level.

```
Ola Larsson: Welcome! How can I assist you today?
Ola Larsson: Please provide topic (e.g., restaurant, transit)
You: weather barcelona tomorrow
Ola Larsson: Please provide the following: location, date
You: barcelona tomorrow
Ola Larsson: Fetch weather for Barcelona on Tomorrow? (yes/no) yes
Ola Larsson: Fetching weather for Barcelona on Tomorrow...
```

We tried solving the system to catching everything in one go, but solving this with the recursive design, proved quite a challenge, and we decided with the limited time, to deem this solution good enough. The whole code for this chatbot version is found in Appendix A.2.

Overall, we believe this approach should simplify the process of extending the system to handle new tasks or details, as we would just create new frames for that type of data abstraction. For instance, we played around with extending the restaurant part of the chatbot, to also be able to book a table, given that it has found a match for a restaurant. This seems rather easy to do now, all we would have to do is extend the RestaurantFrame, with another slot, giving it space for a booking frame (eg. BookingFrame), which could hold something like a name, and a time. This could be accompanied with a servant that, once filled, asks if the users wants to confirm their booking. Starting to fill this BookingFrame out, could be attached as to be started by a servant of the RestaurantFrame.

Furthermore, the concrete example in GUS ([1]) that handles flight bookings, should be quite doable, with this approach, and would be fun further implementation to try.

2.3 Improvements

When it comes to the improvement of our system, there are a range of thing we could do and improve upon. The first improvement that we thought of was to incorporate machine learning, likely transformer-based models or other NLP techniques, to improve intent detection and entity extraction. This would allow the system to understand a broader range of expressions and handle indirect queries more effectively. Implementing more sophisticated natural language processing would allow us to interpret indirect or fragmented responses, reducing reliance on rigid keyword patterns.

Also, implement a dialogue manager that tracks conversational context across multiple turns, enabling the system to handle follow-up questions and maintain coherent dialogues even if the user changes the topic.

Moreover, we could also work on design of the system to allow both the user and the assistant to take more initiative, leading to a more natural conversation flow, as the system is still quite dominant.

Finally, we could use some adaptive learning like feedback loops which allow the system to learn from user interactions. Over time, the assistant could then adapt its responses based on recurring patterns in user behavior. This would likely use Reinforcment learning. Also, utilizing historical interaction data could tailor responses to individual users, which could improve the user experience.

A First implementation

```
import json
import random
import re

with open("valid_values.json", "r") as f:
    valid_data = json.load(f)

# lowercase for case-insensitive matching
valid_locations = [loc.lower() for loc in valid_data["locations"]]
valid_cuisines = [cuisine.lower() for cuisine in valid_data["cuisines"]]
valid_dates = [date.lower() for date in valid_data["dates"]]
valid_times = [time.lower() for time in valid_data["times"]]

chat_bot_name = "Ola Larsson"

KEYWORDS = {
    "weather": ["weather", "forecast", "temperature"],
    "restaurant": ["restaurant", "food", "eat", "dinner", "lunch", "cuisine"],
    "transit": ["bus", "tram", "transit", "train", "metro"]
}

FRAME_CONFIG = {
    "weather": ("Weather", ["location", "date"]),
    "restaurant": ("Restaurant", ["location", "cuisine"]),
    "transit": ("Transit", ["location", "time"]),
}

SLOT_QUESTIONS = {
    "location": "What location? (e.g., " + ",
        ".join(random.sample(valid_data["locations"], 2)) + ")",
    "date": "What date? (e.g., " + ", ".join(random.sample(valid_data["dates"], 2))
        + ")",
    "time": "What time? (e.g., " + ", ".join(random.sample(valid_data["times"], 2))
        + ")",
    "cuisine": "What cuisine? (e.g., " + ",
        ".join(random.sample(valid_data["cuisines"], 2)) + ")"
}


def validate_location(location):
    return location.lower() in valid_locations

def validate_cuisine(cuisine):
    return cuisine.lower() in valid_cuisines

def validate_date(date):
    return date.lower() in valid_dates

def validate_time(time):
    return time.lower() in valid_times

# validators dictionary
validators = {
    "location": (validate_location, lambda: ",
        ".join(random.sample(valid_data["locations"], 2))),
    "cuisine": (validate_cuisine, lambda: ",
        ".join(random.sample(valid_data["cuisines"], 2))),
    "date": (validate_date, lambda: ", ".join(random.sample(valid_data["dates"],
        2))),
    "time": (validate_time, lambda: ", ".join(random.sample(valid_data["times"],
        2)))
}


class Frame:
    def __init__(self, name, slots):
        self.name = name
        self.slots = {slot: None for slot in slots}

    def fill_slot(self, slot, value):
        if slot in self.slots and value:
            self.slots[slot] = value.strip().lower() # store as lowercase

    def is_complete(self):
        return all(value is not None for value in self.slots.values())

    def missing_slots(self):
        return [slot for slot, value in self.slots.items() if value is None]


def create_frame(frame_type):
    config = FRAME_CONFIG[frame_type]
    return Frame(config[0], config[1])


# determine which frame to use based on keywords
def select_frame(user_input):
    text = user_input.lower()
    for key, keywords in KEYWORDS.items():
        if any(keyword in text for keyword in keywords):
            return key
    return None


# extract the first valid value for any slot
def extract_first_valid_value(user_input, slot):
    valid_map = {
        "location": valid_locations,
        "cuisine": valid_cuisines,
        "date": valid_dates,
        "time": valid_times
    }
    user_input_lower = user_input.lower()
    for candidate in valid_map.get(slot, []):
        if re.search(r'\b' + re.escape(candidate) + r'\b', user_input_lower):
            return candidate
    return None


# update current frame with new info from input
def update_frame(user_input, current_frame):
    if current_frame.name == "Weather":
        if current_frame.slots["location"] is None:
            candidate = extract_first_valid_value(user_input, "location")
            if candidate:
                current_frame.fill_slot("location", candidate)
        if current_frame.slots["date"] is None:
            candidate = extract_first_valid_value(user_input, "date")
            if candidate:
                current_frame.fill_slot("date", candidate)
    elif current_frame.name == "Restaurant":
        if current_frame.slots["location"] is None:
            candidate = extract_first_valid_value(user_input, "location")
            if candidate:
                current_frame.fill_slot("location", candidate)
        if current_frame.slots["cuisine"] is None:
            candidate = extract_first_valid_value(user_input, "cuisine")
            if candidate:
                current_frame.fill_slot("cuisine", candidate)
    elif current_frame.name == "Transit":
        if current_frame.slots["location"] is None:
            candidate = extract_first_valid_value(user_input, "location")
            if candidate:
                current_frame.fill_slot("location", candidate)
        if current_frame.slots["time"] is None:
            candidate = extract_first_valid_value(user_input, "time")
            if candidate:
                current_frame.fill_slot("time", candidate)
    return current_frame

def ask_for_missing_info(current_frame):
    missing = current_frame.missing_slots()
    if missing:
        slot = missing[0]
        question = SLOT_QUESTIONS[slot]
        return question, slot
    return None, None

def fill_missing_slots(current_frame, valid_data, validators):
    first_time = True
    while not current_frame.is_complete():
        question, slot = ask_for_missing_info(current_frame)
        if question:
            if first_time:
                print(f"{chat_bot_name}: {question}")
            user_response = input("You: ")
            # generic extraction on the user's response
            candidate = extract_first_valid_value(user_response, slot)
            if candidate is None or not validators[slot][0](candidate):
                examples = validators[slot][1]()
                print(f"{chat_bot_name}: I didn't recognize that {slot}. Please
                    provide a valid {slot} (e.g., {examples}).")
                first_time = False
                continue
            # use extracted candidate for filling the slot.
            current_frame.fill_slot(slot, candidate)
            current_frame = update_frame(user_response, current_frame)
            first_time = True
    return current_frame

def main():
    first_time = True
    while True:
        if first_time:
            print(f"{chat_bot_name}: Welcome to your digital assistant
                {chat_bot_name}. How can I help you?")
        else:
            print(f"{chat_bot_name}: How can I help you?")

        frame_key = None
        while frame_key is None:
            initial_input = input("You: ")
            frame_key = select_frame(initial_input)
            if frame_key is None:
                print(f"{chat_bot_name}: I'm sorry, I couldn't determine your
                     request. Please mention weather, restaurant, or transit.")

        current_frame = create_frame(frame_key)
        current_frame = update_frame(initial_input, current_frame)
        current_frame = fill_missing_slots(current_frame, valid_data, validators)

        if current_frame.name == "Weather":
            print(f"{chat_bot_name}: Fetching weather for
                 {current_frame.slots['location']} on {current_frame.slots['date']}.")
        elif current_frame.name == "Restaurant":
            print(f"{chat_bot_name}: Searching for {current_frame.slots['cuisine']}
                 restaurants near {current_frame.slots['location']}.")
        elif current_frame.name == "Transit":
            print(f"{chat_bot_name}: Retrieving transit info for
                 {current_frame.slots['location']} at {current_frame.slots['time']}.")

        print(f"{chat_bot_name}: Is there anything else I can help you with?
             (yes/no)")
        answer = input("You: ").strip().lower()
        if answer not in ["yes", "y"]:
            print(f"{chat_bot_name}: Goodbye!")
            break

        first_time = False

if __name__ == "__main__":
    main()
```

A.1 Valid values

```
{
    "locations": ["Gothenburg", "Oslo", "Stockholm", "Copenhagen", "Helsinki",
         "Reykjavik", "Paris", "London", "Berlin",
    "Rome", "Madrid", "Lisbon", "Amsterdam", "Brussels", "Vienna", "Prague",
         "Budapest", "Warsaw", "Athens", "Moscow", "Istanbul",
    "Dubai", "New York", "Los Angeles", "Chicago", "Houston", "Miami", "Toronto",
         "Mexico City", "Buenos Aires", "Sao Paulo", "Lima",
    "Santiago", "Cape Town", "Nairobi", "Cairo", "Mumbai", "Delhi", "Bangkok",
         "Singapore", "Kuala Lumpur", "Jakarta", "Beijing", "Shanghai",
    "Tokyo", "Seoul", "Sydney", "Auckland", "Miami", "New Orleans", "Las Vegas",
         "San Francisco", "Seattle", "Vancouver", "Montreal", "Rio de Janeiro",
    "Bogota", "Caracas", "Quito", "Lima", "La Paz", "Santiago", "Buenos Aires",
         "Sao Paulo", "Brasilia", "Lisbon", "Madrid", "Barcelona", "Paris",
    "London", "Dublin", "Amsterdam", "Brussels", "Berlin", "Munich", "Vienna",
         "Prague", "Budapest", "Warsaw", "Moscow", "St. Petersburg", "Istanbul",
    "Athens", "Rome", "Milan", "Florence", "Venice", "Naples", "Copenhagen",
         "Stockholm", "Oslo", "Helsinki", "Reykjavik", "New Delhi", "Mumbai",
         "Bangalore"],
    "cuisines": ["italian", "chinese", "mexican", "indian", "american", "japanese",
         "thai", "french", "spanish", "greek", "turkish", "lebanese",
    "vietnamese", "korean", "peruvian", "brazilian", "argentinian", "chilean",
         "south african", "egyptian", "moroccan", "ethiopian", "nigerian", "kenyan",
    "australian", "new zealand", "british", "german", "swedish", "norwegian",
         "danish", "finnish", "icelandic", "russian", "polish", "czech", "hungarian",
    "austrian", "belgian", "dutch", "portuguese", "swiss", "greek", "turkish",
         "lebanese", "israeli", "palestinian", "syrian", "jordanian", "iraqi",
         "iranian",
    "vegetarian", "vegan", "gluten-free", "lactose-free", "halal", "kosher", "raw",
         "paleo", "low-carb", "low-fat", "low-sugar", "low-sodium", "low-calorie"],
    "dates": ["today", "tomorrow", "this week", "next week", "this month", "next
        month", "this year", "next year",
    "friday", "saturday", "sunday", "monday", "tuesday", "wednesday", "thursday"],
    "times": ["morning", "afternoon", "evening", "night", "midnight", "1 am", "2
        am", "3 am", "4 am", "5 am", "6 am", "7 am", "8 am", "9 am", "10 am",
    "11 am", "12 am", "1 pm", "2 pm", "3 pm", "4 pm", "5 pm", "6 pm", "7 pm", "8
        pm", "9 pm", "10 pm", "11 pm", "12 pm"]
  }
```

A.2 Second implementation

```
import json
import re
import random
from abc import ABC, abstractmethod

# load valid values from the JSON file
with open("valid_values.json", "r") as f:
    valid_data = json.load(f)

# define the chat bot name
chat_bot_name = "Ola Larsson"

# define some keywords for each topic
KEYWORDS = {
    "weather": ["weather", "forecast", "temperature"],
    "restaurant": ["restaurant", "food", "eat", "dinner", "lunch", "cuisine"],
    "transit": ["bus", "tram", "transit", "train", "metro"]
}

# function to extract topic from text
def extract_topic(text, valid_values):
    text = text.lower()
    matches = []
    for topic, words in KEYWORDS.items():
        for word in words:
            m = re.search(r'\b' + re.escape(word.lower()) + r'\b', text)
            if m:
                matches.append((m.start(), topic))
    if matches:
        matches.sort(key=lambda x: x[0])
        return matches[0][1]
    return ""

# function to extract a value from text
# uses the valid values to match the text
def extract_value(text, valid_values):
    text = text.lower()
    for item in valid_values:
        if re.search(r'\b' + re.escape(item.lower()) + r'\b', text):
            return item.lower()
    return ""

# function to validate a value against a list of valid values
def validate_value(value, valid_values):
    return value.lower() in [v.lower() for v in valid_values]


# define the frame structure for the chatbot
# idea is to have a tree-like structure of frames, where each frame can have slots
# and each slot is a frame itself
# either a leaf-frame with a value or a non-leaf-frame with slots
class Frame(ABC):
    def __init__(self, name, valid_values=None, extract=None, validate=None):
        self.name = name
        self.value = None
        self.valid_values = valid_values
        self.extract = extract
        self.validate = validate
        self.slots = {}
        self.servants = []

    # check if the frame is a leaf
    def is_leaf(self):
        return not self.slots

    # check if the frame is complete
    # a frame is complete if it is a leaf and has a value
    def is_complete(self):
        if self.is_leaf():
            return self.value is not None
        else:
            return all(slot.is_complete() for slot in self.slots.values())

    # recursive function to fill frame and its slots with value extracted from the
        text
    # if the frame is a leaf, it tries to extract a value from the text and
        validate it against the valid values
    # if the frame is not a leaf, it tries to fill its slots
    def fill(self, text):
        if self.is_leaf():
            if self.value is None:
                candidate = self.extract(text, self.valid_values)
                if candidate and self.validate(candidate, self.valid_values):
                    self.value = candidate
                    # if the frame is a leaf and we have a value, we are done here
                    return True
            return False
        else:
            for slot in self.slots.values():
                slot.fill(text)
            # if all slots are complete, we can try to run the frame
            return self.is_complete()

    # this is our 'demon' for a frame
    # if the frame is not complete, it will prompt the user to provide the missing
        values
    # if the frame is a leaf, it will prompt the user to provide the value
    # else it will prompt the user to provide values for the missing slots
    # meaning it tries to fill several slots at once, on the same level
    def prompt(self):
        if self.is_leaf():
            while self.value is None:
                examples = ', '.join(random.sample(self.valid_values, min(2,
                    len(self.valid_values))))
                user_input = input(f"{chat_bot_name}: Please provide {self.name}
                    (e.g., {examples}) \nYou: ")
                if not self.fill(user_input):
                    print(f"{chat_bot_name}: Invalid {self.name}. Valid options: {',
                        '.join(self.valid_values.take(5))}.")
        else:
            missing = [name for name, slot in self.slots.items() if not
                slot.is_complete()]
            if missing:
                prompt_msg = f"{chat_bot_name}: Please provide the following: " + ",
                     ".join(missing) +"\nYou: "
                user_input = input(prompt_msg)
                for name in missing:
                    self.slots[name].fill(user_input)
                for slot in self.slots.values():
                    if not slot.is_complete():
                        slot.run(user_input)

    # method for running the frame
    # it will try to fill the frame with the initial text
    # if the frame is not complete, it will prompt the user
    # if the frame is complete, it will run the attached servants
    def run(self, initial_text):
        self.fill(initial_text)
        if not self.is_complete():
            self.prompt()
        for servant in self.servants:
            servant(self)

# frame for location
# it has a list of valid locations
class LocationFrame(Frame):
    def __init__(self):
        valid = [loc.lower() for loc in valid_data["locations"]]
        super().__init__("location", valid_values=valid, extract=extract_value,
             validate=validate_value)

# frame for date
# it has a list of valid dates
class DateFrame(Frame):
    def __init__(self):
        valid = [date.lower() for date in valid_data["dates"]]
        super().__init__("date", valid_values=valid, extract=extract_value,
             validate=validate_value)

# frame for time
# it has a list of valid times
class TimeFrame(Frame):
    def __init__(self):
        valid = [time.lower() for time in valid_data["times"]]
        super().__init__("time", valid_values=valid, extract=extract_value,
             validate=validate_value)

# frame for cuisine
# it has a list of valid cuisines
class CuisineFrame(Frame):
    def __init__(self):
        valid = [cuisine.lower() for cuisine in valid_data["cuisines"]]
        super().__init__("cuisine", valid_values=valid, extract=extract_value,
             validate=validate_value)

# frame for weather
# it has a location and date slot
# it has a servant that will fetch the weather, if the user asks for it
class WeatherFrame(Frame):
    def __init__(self):
        super().__init__("weather")
        self.slots["location"] = LocationFrame()
        self.slots["date"] = DateFrame()
        self.servants.append(self.weather_servant)
    def weather_servant(self, frame):
        loc = frame.slots["location"].value.capitalize()
        date = frame.slots["date"].value.capitalize()
        answer = input(f"{chat_bot_name}: Fetch weather for {loc} on {date}?
            (yes/no) ")
        if answer.lower() == "yes":
            print(f"{chat_bot_name}: Fetching weather for {loc} on {date}...")

# frame for transit
# it has a location and time slot
# it has a servant that will find transit options, if the user asks for it
class TransitFrame(Frame):
    def __init__(self):
        super().__init__("transit")
        self.slots["location"] = LocationFrame()
        self.slots["time"] = TimeFrame()
        self.servants.append(self.transit_servant)

    def transit_servant(self, frame):
        loc = frame.slots["location"].value.capitalize()
        time_val = frame.slots["time"].value.capitalize()
        answer = input(f"{chat_bot_name}: Find transit options for {loc} at
            {time_val}? (yes/no) ")
        if answer.lower() == "yes":
            print(f"{chat_bot_name}: Finding transit options for {loc} at
                {time_val}...")

# frame for restaurant
# it has a location and cuisine slot
# it has a servant that will find restaurants, if the user asks for it
class RestaurantFrame(Frame):
    def __init__(self):
        super().__init__("restaurant")
        self.slots["location"] = LocationFrame()
        self.slots["cuisine"] = CuisineFrame()
        self.servants.append(self.restaurant_servant)

    def restaurant_servant(self, frame):
        loc = frame.slots["location"].value.capitalize()
        cuisine = frame.slots["cuisine"].value.capitalize()
        answer = input(f"{chat_bot_name}: Search {cuisine} restaurants in {loc}
            (yes/no) ")
        if answer.lower() == "yes":
            print(f"{chat_bot_name}: Searching {cuisine} restaurants in {loc}...")

# defines a frame for picking a topic to discuss
# this is like the dialog frame in GUS
class TopicFrame(Frame):
    def __init__(self):
        valid = [topic.lower() for topic in valid_data["topics"]]
        super().__init__("topic", valid_values=valid, extract=extract_topic,
            validate=validate_value)
        self.topic_to_frame = {
            "weather": WeatherFrame,
            "transit": TransitFrame,
            "restaurant": RestaurantFrame
        }

    # get the frame for the selected topic
    def get_frame(self):
        return self.topic_to_frame[self.value]() if self.value in
            self.topic_to_frame else None

# main for running the chatbot
# it will run the topic frame which prompts the user to select a topic
# then it will run the selected frame, recursively prompting the user for missing
    values if needed
def main():
    print(f"{chat_bot_name}: Welcome! How can I assist you today?")
    while True:
        topic_frame = TopicFrame()
        topic_frame.run("")
        frame = topic_frame.get_frame()
        frame.run("")
        cont = input(f"{chat_bot_name}: Would you like to continue? (yes/no)
            ").strip().lower()
        if cont == "no":
            print(f"{chat_bot_name}: Goodbye!")
            break
        else:
            if cont == "yes":
                print(f"{chat_bot_name}: How can I assist you next?")

if __name__ == "__main__":
    main()
```

References

[1] D. G. Bobrow, R. M. Kaplan, M. Kay, D. A. Norman, H. Thompson, and T. Winograd, "Gus, a frame-driven dialog system," *Artificial intelligence*, vol. 8, no. 2, pp. 155–173, 1977.