Emotional responsive interaction with Pepper

Luca Corvitto **Lorenzo Faiella**

1835668 **1835950**

[corvitto.1835668, faiella.1835950]@studenti.uniroma1.it

Elective in Artificial Intelligence

Human Robot Interaction & Reasoning Agents

Artificial Intelligence and Robotics

Contents

Chapter 1

Introduction

All the authors contributed equally to the project.

Figure 1.1: Report of World Robotics 2023.

In the 21st century, robots have brought huge changes in the way we live, work, and interact with the world. We are still in the early stages of this revolution, but robots are already indispensable in our daily lives. This growth is proceeding at an unprecedented rate as shown

in the last report of *World Robotics 2023* in Figure 1.1: just in 2022 there have been 553.052 new industrial robots installations around the world. Machines have transcended their traditional roles in manufacturing and assembly lines, arriving in fields such as healthcare, education, entertainment, and even our homes. Social robots are the most suited to accomplish all these tasks outside factories. They are a particular type of robot that interacts and communicates with humans, and most of them, to achieve some empathy are designed with a screen to represent the head or "face" to dynamically communicate with users or some features that resemble human appearances and gestures. Social robots with their design to respond to human emotions are perfect in healthcare settings such as assisting therapists or in education as interactive tutors that adapt their unique teaching style to every student.

One of the most famous social robots is *Pepper*, developed by *Softbank robotics*, Figure 1.2 shows its appearance and its captivating design. For our project, we will be using Pepper to simulate an interview with a user where the robot shows emotive reactions to create a deeper bond with the human and then it will test the user's attention on a little fun quiz. We chose to carry out our experiments with this robot, firstly because it has an open and fully programmable platform, and also because of some of its main features, which are here listed:

- 20 degrees of freedom for natural and expressive movements.
- Speech recognition and dialogue available in 15 languages.
- Perception modules to recognize and interact with the person talking to him.
- Touch sensors, LEDs and microphones for multimodal interactions.
- Infrared sensors, bumpers, an inertial unit, 2D and 3D cameras, and sonars for omnidirectional and autonomous navigation.

Pepper with its 120cm of height has no trouble perceiving his environment and entering into a conversation when it sees a person, the touch screen on his chest displays content to highlight messages and support speech, and his curvy design ensures danger-free use and a high level of acceptance by users. Our project wants to present a simplistic example of how a housing robot initialization conversation should be handled. We were inspired by the work carried out by Kraus et al. 2022, starting from their effort, we extend it by adding emotive physical reactions during the human-robot interaction to make more natural and less alienating conversations. Pepper exchanges personal information with the user by telling more about itself to put the

human at ease, making the interaction more like a real dialogue rather than a formal interview. After collecting information in the initial dialogue, the robot poses a quiz to test the user's attention during the previous conversation, and Pepper's mood will change according to the given answers. This choice was made to experiment with the robot's behavior adaptation in different scenarios, thus simulating a social robot that wants to create sentimental bonds with humans.

Figure 1.2: 100 Pepper cheerleading squad winning Guinness World Records certificate.

Chapter 2

Related Works

In the following sections, we will dive into the relevant papers and concepts of HRI and RA that we used as the foundation for our work.

2.1 Human Robot Interaction

HRI and research on it represent a multidisciplinary field, it involves "the study of the humans, robots, and the ways they influence each other" (Fong et al. 2003). The domain of social robotics is the one that interests us the most for this project. These types of robots "exist primarily to interact with people" (Kirby et al. 2010) or arouse social responses from them. Sharing a similar morphology with the users, they can communicate in a manner that supports the natural communication modalities of humans, examples include facial expression, body posture, gesture, gaze direction, and voice. In a classical S-O-R (Stimulus-Organism-Response) paradigm, a stimulus can include expressions of other people's internal states (Eroglu et al. 2001); so, internal emotions, even the robot ones, offer important stimuli during human-robot interactions. Artificial emotions can be generated with two different approaches, the first one is the static approach where the robot's animation is manually coded pose-to-pose (our approach). The second approach is the dynamic one, which can be either proactive or reactive; proactive emotion generation may be inspired by graphic animation design, such as Disney's 12 basic principles of animation (Ghani and Ishak 2012), while, reactive emotion generation relies on data generated through the recognition of human emotions in general, a commonly used technique consists in a direct imitation by tracking human emotional expressions (Matsui et al. 2005, Yoon

et al. 2018). The survey by Stock-Homburg 2022, after analyzing many research papers, states that robots can be programmed to express emotions, despite not actually having them, and humans can recognize and accept them. Emotions are so important that studies, such as Gockley et al. 2006, underline the fact that a robot expressing positive emotions is more accepted, and people who evaluate a robot positively also express more interest in the interaction.

In our project, taking inspiration from Kraus et al. 2022, we propose a simplified version of robot emotions, instead of using the six primary emotions we only use happiness and sadness, we believe that these two emotions, combined with body gestures, in our type of interaction are enough to better engage the user in a conversation.

2.2 Reasoning Agents

Dealing with people in the real world can be really hard to handle, mainly because of the unpredictability of human actions. In our case, the user can have various levels of robot acceptance, and his behavior and answers can change according to this fact, so the robot must be able to change the approach and dialogue lines to better accommodate the different types of humans it is dealing with. The solution to this problem can be planning, more specifically non-deterministic planning, where actions are uncertain before execution time. In our case, since we are dealing with a robot, we speak of *contingent planning* because the environment is observable through sensors, that can be faulty, so the agent acts under incomplete information. In *conditional planning*, the effects of actions are contingent on the outcomes of previous actions, while the execution is not; on the other hand , in *contingent planning* Majercik and Littman 2003 both the effects and execution of actions are contingent on the outcomes of previous actions.

In our work, we adopted a special type of PDDL planning that can adopt non-deterministic planning with particular constructs, this will be explored in more detail in section 3.2.

Chapter 3

Solution

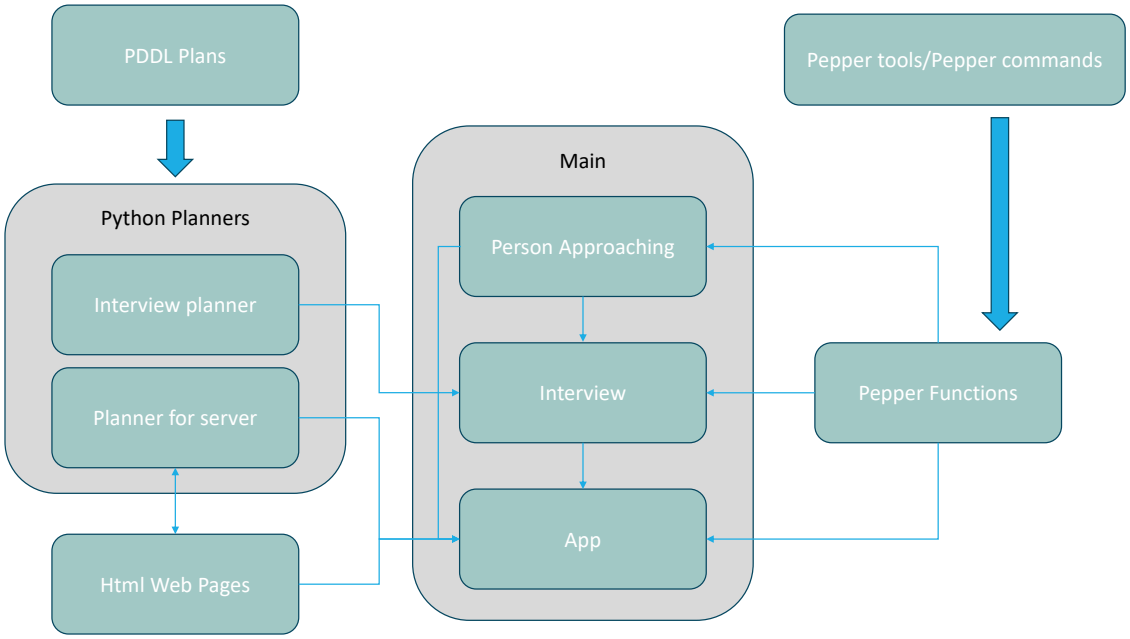

Figure 3.1: Scheme of the solution's architecture.

The architecture of the solution, as can be seen in Figure 3.1, is fairly simple. Everything is handled by the `main`, and it exploits the robot's functions such as **`Initialization`**, **`Move`**, **`Dialogue`**, etc... The `main` also controls the tablet interaction, which in our case we implemented as an HTML server, through the `app` module. The PDDL plans are `.txt` files generated by the planner that will be further explored in Section 3.2, the planner executors are also called by the `main` when needed to guide the interview and the quiz on the tablet. `Person Approaching` handles the start of the interaction with the sonar and all the specific gestures performed by

Pepper are manually designed and programmed to meet our emotive reaction requirements. Our entire program can run on a single terminal, but to interact with the robot we use another terminal to use Pepper tools to send signals or simulate the conversation.

3.1 Human Robot Interaction

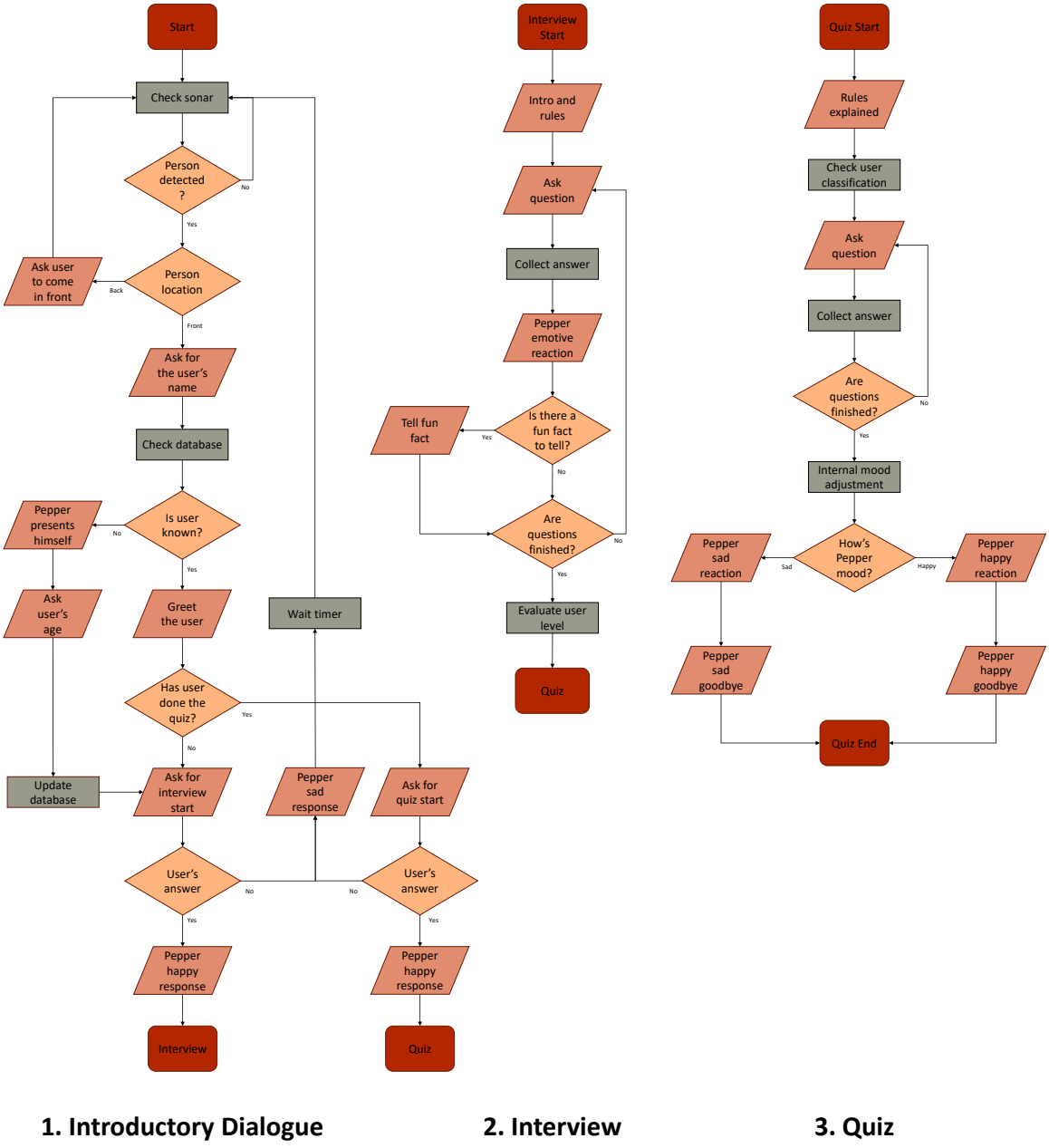

Figure 3.2: Flow charts of the solution's architecture.

The flow charts in Figure 3.2 summarize the entire human-pepper interaction. The robot during its inactivity period is always looking for people with its sonar; whenever a person is noticed,

the robot checks for its position, if the person is behind Pepper it asks the user to come in front of it and, when the human being is in front, the real interaction starts. Pepper starts interacting by asking for the user's name; after receiving an answer, it checks the database to know if it is the first time that it has met the person; if the user is unknown, the robot presents itself, and then asks for some personal info and updates its internal database; if the user is already known, they is greeted by Pepper and then asked to take the quiz again if it has already been taken. In both cases in which the user has never taken the interview before, Pepper asks if they wants to start an interview, if the answer is positive the robot reacts positively and then the interview starts immediately, on the other hand, if the answer is negative, the robot responds politely showing a sad attitude and then, after waiting a bit, the interaction starts from the beginning with the sonar check. Pepper begins the interview with an introduction and an explanation of the rules, then it starts asking some questions and telling some fun facts about itself, after collecting all the answers Pepper will compute the user's level of robot approval. To achieve a greater bond with the robot and improve the quality of the interaction, the robot will show its mood with an emotive reaction after every answer, an emotional response that is dependent on the answer received: an answer that shows disapproval of robots will lead to a sad reaction, a positive one will lead to a happy reaction. At the end of the interview, Pepper presents to the human being a quiz to test their attention on the fun facts, some questions can change depending on the user's age, and the level of the quiz is chosen according to the level of robot approval determined by the previous phase of interaction, the lower the level, the easier the quiz will be. Once all questions presented by the robot are answered, an internal mood adjustment is executed according to the user's level, more precisely: the quiz is considered passed with 2 right answers if the difficulty was easy, 3 right answers are required to pass a medium quiz, and 4 right ones to pass a hard quiz. The robot performs a happy or sad reaction at the end of the quiz depending on the quiz results, then it says goodbye to the user.

To better simulate a human-robot interaction we implemented an HTML server as a tablet representation where communication between the user and Pepper is as easy and immediate as possible. Usually, an interaction with a robot in a single small session isn't enough to create a solid bond, so we decided to include emotive reactions to facilitate the emotive connection with the robot and maximize the bonding over the experience.

Adding the emotional reaction to the data collection phase, we make the interaction more

natural and we put the user in a condition of greater empathy with the robot, meanwhile, providing information about the robot, we make the user more inclined to provide their own information as in a natural dialogue between humans.

3.2 Reasoning Agents

All the robot's reasoning abilities are implemented using the Planning Domain Definition Language (PDDL), more precisely our architecture exploits two generated plans, the first one handles the interview and the categorization of the user's level of robot approval, and the second one controls the quiz by choosing the questions to show according to the user level and evaluates all the answers to determine the robot's mood. Our tasks require a special type of reasoning because interacting with a user in real-time implicates non-deterministic actions and partial observability. Contingent-FF proposed by Hoffmann and Brafman 2006, is the planner that meets our requirements, it is a modification of the Fast-Forward (FF) planner that enables Contingent Planning. This particular language implements the `(:observe (<pred>))` construct that permits variables with dynamic values that can change during the plan execution, the variables that can be affected by this special construct are defined with the `(unknown (<pred>))` construct that formally represents a partially observable variable. Contingent-FF generates a plan with a tree structure where branches are created when observations of unknown variables are made, and when executing the plan in real-time the solution follows a single branch, so it computes all possible solutions and only one takes place in the real world. As we can see from the code below, some actions have a `TRUESON` and a `FALSESON` that represent the branch leaves when making an observation, in this case, different sons are created when the `CHECK_ANSWER` actions are performed.

Plan of the hard quiz:

```
------------------------------------------------
  0||0 --- QUIZ_START --- SON: 1||0
------------------------------------------------
  1||0 --- SHOW_QUESTION QUESTION_MARS --- SON: 2||0
------------------------------------------------
  2||0 --- SHOW_QUESTION QUESTION_BASEBALL --- SON: 3||0
------------------------------------------------
  3||0 --- SHOW_QUESTION QUESTION_SWARM --- SON: 4||0
------------------------------------------------
  4||0 --- SHOW_QUESTION QUESTION_1 --- SON: 5||0
```

```
-------------------------------------------------
  5||0 --- CHECK_ANSWER QUESTION_MARS --- TRUESON: 6||0 --- FALSESON: 6||1
-------------------------------------------------
  6||0 --- CHECK_ANSWER QUESTION_BASEBALL --- TRUESON: 7||0 --- FALSESON: 7||1
  6||1 --- SAD_EMOTIONAL_RESPONSE_HARD QUESTION_MARS QUESTION_1 QUESTION_BASEBALL QUESTION_SWARM --- SON: 7||-1
-------------------------------------------------
  7||0 --- CHECK_ANSWER QUESTION_SWARM --- TRUESON: 8||0 --- FALSESON: 8||1
  7||1 --- SAD_EMOTIONAL_RESPONSE_HARD QUESTION_BASEBALL QUESTION_1 QUESTION_MARS QUESTION_SWARM --- SON: 8||-1
-------------------------------------------------
  8||0 --- CHECK_ANSWER QUESTION_1 --- TRUESON: 9||0 --- FALSESON: 9||1
  8||1 --- SAD_EMOTIONAL_RESPONSE_HARD QUESTION_SWARM QUESTION_1 QUESTION_MARS QUESTION_BASEBALL --- SON: 9||-1
-------------------------------------------------
  9||0 --- HAPPY_EMOTIONAL_RESPONSE_HARD QUESTION_1 QUESTION_SWARM QUESTION_MARS QUESTION_BASEBALL --- SON: 10||-1
  9||1 --- SAD_EMOTIONAL_RESPONSE_HARD QUESTION_1 QUESTION_SWARM QUESTION_MARS QUESTION_BASEBALL --- SON: 10||-1
-------------------------------------------------
```

Chapter 4

Implementation

The project was implemented in the Ubuntu operating system, through Docker, using the docker image provided for the Pepper robot. The code was written mainly in Python, but PDDL, HTML, CSS and Javascript were also used for more specific tasks. We tested and carried out the simulation of the overall interaction on the Choregraphe Pot et al. 2009 simulator and on the Firefox browser. The code is available on github.

4.1 Human Robot Interaction

In this section we will describe into details how the main blocks shown in 3 were implemented. To do it is fundamental to explain the `pepper_functions` module, that contains classes and functions used across the entire interaction.

4.1.1 Pepper functions

This module manages the basic commands of the robot to make it remember, move, speak, sense, and react emotionally. To do so, we created several classes:

`class Memory()`: allows to save, load and update users data in `users_data.txt`, and the data useful for the interaction of the current user in `current_user.txt`.

`class Initialization()`: uses the `pepper_cmd` module to initialize the robot in an alive behavior, starting the `ALBackgroundMovement`, `ALBasicAwareness` and `ALSpeakingMovement` services.

`class Move()`: handles every possible movement of the robot. In our case, it is used to perform the movements needed for the emotive reaction of the robot.

```
def change_posture(self, joints, values, sleeping_time = 0.5, stiffness
    =0.0):
     joint_list = []
     joint_values = []
     posture = pepper_cmd.robot.getPosture()
     for i, (j, v) in enumerate(zip(joints, values)):
       # input v is in degrees so I will turn into radians
       joint_values.append(v*np.pi/180)
       joint_list.append(self.jointNames[self.InputJoints[j]])
     pepper_cmd.robot.motion_service.angleInterpolation(joint_list,
    joint_values,1.0,True)
     if stiffness > 0.0:
       end_time = time.time() + sleeping_time
       while time.time() < end_time:
         pepper_cmd.robot.motion_service.angleInterpolation(joint_list,
    joint_values,1.0,True)
     else:
       time.sleep(sleeping_time)
     print("Posture changed in joints: {}".format(joint_list))

     return posture
```

Through the `pepper_cmd` module the **`change_posture`** function changes the angle orientation of the joints allowing the robot to take feasible postures. Setting the stiff value to `True` blocks the robot joints to the specified position for the specified sleeping time.

`class Dialogue()`: allows the robot to speak, through the **`say`** function that uses the `ALAnimatedSpeech` service to convert text to speech accompanied by contextual gestures, and to "listen", through the **`listen`** function, which waits patiently for an answer from the user, using the `ALSpeechRecognition` service integrated into the Pepper robot.

```
def listen(self, timeout = 30, what_requires = 'aything'):
```

```
2
3     if what_requires == 'answer': #interview
4       answer = pepper_cmd.robot.asr(vocabulary = self.ans_vocab, timeout =
      timeout)
5
6       while answer not in self.ans_vocab:
7         if not answer:
8           answer = self.say(sentence = "Sorry, I did not hear you, repeat
      please.", answer_time_window = 30)
9         else:
10          answer = self.say(sentence= "Sorry for my lack of comprehension,
      but I need you to answer with just 'yes', 'no' or 'maybe'.",
      answer_time_window = 30)
11
12    elif what_requires == 'age': #requesting age
13      answer = pepper_cmd.robot.asr(vocabulary = self.age_vocab, timeout =
      timeout)
14
15      while answer not in self.age_vocab:
16        if not answer:
17          answer = self.say(sentence = "Sorry, I did not hear you, repeat
      please.", answer_time_window = 30)
18        else:
19          # check if user's age is an integer
20          try:
21            if int(answer):
22              in_answer = self.say(sentence = "Are you sure {} is your age?
       Please answer with just yes or no.".format(answer), answer_time_window
      = 30)
23              if in_answer == 'yes':
24                self.say(sentence='Uh, okay then.')
25                return answer
```

```
26                else:
27                  answer = self.say(sentence= 'So, what is really your age?',
      answer_time_window = 30)
28              except:
29                answer = self.say(sentence= 'Sorry, but I need you to answer me
      with an integer number.', answer_time_window = 30)
30
31      else: # requesting name and other cases
32        answer = pepper_cmd.robot.asr(vocabulary= self.default_vocab, timeout
      = timeout)
33
34        while not answer:
35          answer = self.say(sentence = "Sorry, I did not hear you, repeat
      please.", answer_time_window = 30)
36
37      return answer
```

The **listen** function handles all the possible type of answers and interaction cases of the entire interaction. In the case in which the robot is asking something that requires an affirmative or a negative answer (to be more accurate during the interview we extended the possible answer to 3 in order to have a 3-point Likert scale of answers), it refuses every response different from the expected one; in the case of asking the age it accept every integer number inside a suitable range of ages, meanwhile in the case of asking the names it accepts every response.

class Sensor(): allows the robot to sense its surroundings and detect every person trying to approach it. Since Pepper is equipped with two sensors, one frontal sonar and one on its back, we also handled the case in which the user tries to approach the robot from behind.

```
1 def sense(self, approaching = False, T = 1.0, start_time=0.0, wait_time
    =4.0):
2     pepper_cmd.robot.startSensorMonitor() #start sensing
3     if not approaching:
4         print("Waiting for signal...")
5         try:
```

```
6             while not approaching:
7                 sensed = pepper_cmd.robot.sensorvalue() #value sensed
8                 approaching = (0.0 < sensed[1] < T) or (0.0 < sensed[2] < T
    )
9                 start_time = time.time()
10        except KeyboardInterrupt:
11            pepper_cmd.robot.stopSensorMonitor()
12            sys.exit(0)
13        return self.sense(True, 1.0, start_time, 4.0)
14    else:
15        print("Person approaching...")
16        while approaching:
17            sensed = pepper_cmd.robot.sensorvalue()
18            front = (0.0 < sensed[1] < T)
19            back = (0.0 < sensed[2] < T)
20            approaching = front or back
21            if time.time() - start_time >= wait_time:
22                print("Person waiting {}.".format("front" if front else "
    back"))
23                if back:
24                    # wait for person to leave
25                    while(back):
26                        if self.count == 0:
27                            self.dialogue.say("Please come in front of me,
    I feel a bit uneasy if you stay behind me while we talk.")
28                            self.count+=1
29                        else:
30                            self.dialogue.say("Please come in front of me."
    )
31                        time.sleep(3.0)
32                        sensed = pepper_cmd.robot.sensorvalue()
33                        back = (0.0 < sensed[2] < T)
```

```
34                 if front:
35                     if self.count == 0:
36                         self.dialogue.say("Hello!")
37                     else:
38                         self.dialogue.say("Hello! Thank you for accepting
    my request!")
39                         self.count=0
40                     pepper_cmd.robot.stopSensorMonitor() #stop sensing
41                     return True
42             else:
43                 front = False
44                 back = False
45         if not front:
46             print("Person approached, but then left.")
47             return self.sense(False, 1.0, 0.0, 4.0)
```

The **sense** function starts the sensors equipped to the robot and waits until the monitored values reach a threshold for a declared amount of time, specified by the `wait_time` parameter. We choose a suitable time and threshold distance `T` meters from the robot to avoid interaction with people passing by. In the case that the user is detected behind the robot, it uses the **say** function from the **Dialogue** class to ask the user to move to its front, and it stays in this loop until the frontal sensors detect an attempted approach.

`class` **Emotion**(): handles the most important part of our solution, that is, the emotional reaction of the Pepper robot to the user input. It is composed by the following functions:

```
def reaction(self, user_response, mode="interview"):
    print(user_response)
    if user_response >= 1:
      print("Positive emotive reaction")
      if mode == "interview":
        self.dialogue.say("Yeee.")
      elif mode == 'quiz':
        self.dialogue.say("Wow! You did remember a lot about me, thanks!")
```

```
9         self.move.change_posture(["LWY","RWY","LSP","RSP"],[-90,90,40,40],
    sleeping_time=0.1)
10        self.move.change_posture(["LWY","RWY","LSP","RSP","LER","RER"
    ],[-90,90,90,90,-90,90],sleeping_time=0.5)
11        self.rp_service.goToPosture('Stand',1.0) # come back to initial
    position
12      elif user_response == 0:
13        print("Neutral emotive reaction")
14        self.dialogue.say("Understood.")
15      else:
16        print("Negative emotive reaction")
17        if mode == "interview":
18          self.dialogue.say("Oh.")
19        elif mode == 'quiz':
20          self.dialogue.say("Oh, you didn't pay attention when I was speaking
    , did you?")
21        end_time = time.time() + 1
22        while time.time() < end_time:
23          self.rp_service.goToPosture('Crouch',1.0)
24        #time.sleep(1.0)
25        self.rp_service.goToPosture('Stand',1.0)
26
27      if mode == "quiz":
28        if user_response >=0:
29          self.dialogue.say("Thank you for your time, I hope you enjoyed the
    time spent together as much as I did! Until next time!")
30          self.move.change_posture(["LWY","RWY","LSP","RSP"],[-90,90,40,40],
    sleeping_time=0.1)
31          self.move.change_posture(["LWY","RWY","LSP","RSP","LER","RER"
    ],[-90,90,90,90,-90,90],sleeping_time=0.5)
32          self.rp_service.goToPosture('Stand',1.0)
33        else:
```

```
34          self.dialogue.say("Thank you for your time, bye.")
35          end_time = time.time() + 1
36          while time.time() < end_time:
37            self.rp_service.goToPosture('Crouch',1.0)
38          self.rp_service.goToPosture('Stand',1.0)
39
40      return
41
42  def happy_emotional_response(self, mode):
43      self.reaction(1,mode=mode)
44      return
45
46  def sad_emotional_response(self,mode):
47      self.reaction(-1,mode=mode)
48      return
49
```

The **reaction** function is the core of this class, it handles the emotive reactions of the robot along all the interaction, and it changes slightly depending on the phase of interaction in which its called, managed by the `mode` parameter.

During the interview phase this function handles three types of `user_response`, that are the three types of response it accepts: `"yes"`, `"no"`, and `"maybe"`. In the *neutral* case the robot do not perform any sort of emotive reaction, since the user stays vague. In the *positive* case it uses the **change_posture** function from the **Move()** class, while in the *negative* case it uses the `goToPosture` function from the `ALRobotPosture` service to express its emotion. In both cases, after the action is performed the robot comes back to its initial position.

Finally, during the final phase of the interaction, the robot greets the user expressing for the last time its emotion over the overall interaction with them.

The last two functions, **happy_emotional_response** and **sad_emotional_response**, call the **reaction** function, handling the *positive* and *negative* cases and are used in the first and last phases to make the code more readable.

4.1.2 Person approaching

This block manages the first phase of the interaction, in which the robot patiently awaits an attempt of engagement from the user and subsequently gets to know them.

Here we uses all of the classes defined in 4.1.1, directly or indirectly. The function **interact** covers this entire first phase; it is a recursive function that calls itself every time the user decides not to go on with the interaction. In this way, the robot continues to stay in a waiting state, ready to interact again with any user tries to approach it.

interact uses the **sense** function from the **Sensor** class to check the user approach, after that it takes advantage of functions from both the **Dialogue** and **Memory** classes, together with external textual files, to check and/or update its database for user information.

For the last part of this first phase, if the user has done the quiz previously, **interact** uses both the **say** and **listen** functions of the **Dialogue** class and **sad_emotional_response** or **happy_emotional_response** of the **Emotion** class to react accordingly to the user decision; otherwise, if the user has not done the quiz yet, it simply uses the **Dialogue** class to ask the user if they want to continue the interaction with the next phase.

When this function ends, it returns two outputs:

- information about the current user;
- a Boolean variable that will be used to manage the next interaction performed by the main module.

4.1.3 Interview

In this block we perform the second and maybe the most important phase of this human-robot interaction.

This module uses functions from the **Dialogue** and **Emotion** classes. It is made up of three functions: **ask_questions**, the main one, **emotional_reaction**, and **compute_user_level**.

ask_questions is an iterative function that iterates over the list of questions and fun facts the robot proposes to the user to retrieve some kind of information. In our case, it asks for the user's opinion about robots, robot usage, and robotization. Questions and answers are managed through the **Dialogue** class functions, **say** and **listen**. After each user's answer, **ask_questions**

saves it in a list to process all the information at the end.

emotional_reaction processes each answer and then call the **reaction** function from the **Emotion** class to express the emotional reaction of Pepper to the user's opinion.

compute_user_level, instead, processes the input list summing the value of each answer, that could be 0, if the answer was "maybe", 1, if the answer expresses a favorable opinion about robots, that depends on the question posed, and -1 otherwise.

4.1.4 Quiz

This last block performs the final phase of the interaction, that is the tablet interaction.

Here we use the Flask framework (Grinberg 2018), to create a server-client connection between the terminal and the robot simulator. To emulate the tablet we use several HTML templates running on the Firefox browser. We designed two different styles, one for the web pages containing the questions with multiple answers, and one for the web pages communicating a right/wrong answer and the start or the end of the quiz. Exploiting Flask functionalities we defined the style inside the `index.html` file and then extended it to all the other templates.

The quiz is set as a sequence of 4 multiple-choice questions, where the user has to click on the appropriate answer. The click is managed by the script of each HTML file, since the position of the correct answer differs every time. All of this starts with the call of the `start_server` function that run the `app` module. It exploits Flask properties and functionalities to call different functions according to the current URL.

```
1 @app.route("/")
2 def home():
3     update_info()
4     return render_template("quiz_start.html", name=user_info['user_name'])
```

For example, for the root URL, the module defines a function **home** that updates the info on the user through the **Memory** class's functions, and then render the HTML page specified in the input of the `render_template` function from the flask library.

next_question is one of the core functions, that handles the entire quiz according to the user level computed in 4.1.3, and to the commands of the planner, that we'll explain better in 4.2.2.

start_evaluation is an other core function, called at the end of the quiz, that evaluates the user score and then, through **happy_emotional_response** and **sad_emotional_response** from the **Emotion** class directs the robot to perform its last emotional reaction, followed by its farewell.

4.2 Reasoning Agents

Using Contingent-FF we were able to solve the uncertainty problem and model non-deterministic complication by introducing unknown predicates and observation actions. We have decided to create two different PDDL domains, firstly because it makes our code much more readable and modular, and secondly, because a single domain would have been too computationally expensive since non-deterministic planning can run in exponential time, thus generating a tree with too many branches.

4.2.1 Interview

Regarding the interview, we produced a single plan, where the structure of the whole interaction is defined as a sequence of actions. All actions take the `?int - interview` parameters as input and the `interview_user` action. Contingent planning is used to classify the user according to his or her answers that aren't known beforehand, the types and predicates of this domain are shown in the following snippet.

```
(:types
    interview
)

(:predicates
    (difficulty_medium)      ; true if the quiz difficulty is set to medium
    (difficulty_hard)      ; true if the quiz difficulty is set to hard

    (interview_done ?int - interview)      ; true if the personal interview has been completed

    (checked_hard)
    (checked_medium)
    (classification_done)
)
```

Here follows a brief table with descriptions of all the actions of the interview domain.

Table 4.1: Description of the interview_domain actions.

Action	Description
interview_user	Ask questions and tell fun facts about robots
check_user_interview_hard	Check if user is classified as robot lover
check_user_interview_medium	Check if user is classified as robot appreciator
classify_hard_true	Set quiz level as hard for the user
classify_hard_false	Confirm that user level is not hard
classify_medium_true	Set quiz level as medium for the user
classify_medium_false	Confirm that user level is not medium
classify_easy_true	Set quiz level as easy for the user

The `interview_planner` module manages the entire interview phase. Reading the PDDL plan it takes different actions according to it, and these actions 4.1 call different functions of the `interview` module.

The `interview_user` action is taken to make the robot starts with the second phase of the interaction, calling the **`ask_questions`** function, while the `check_user_interview` actions make use of the value calculated by the **`compute_user_level`** function to classify the user level for the following quiz.

4.2.2 Quiz

Pepper domain handles the quiz questions to show on the tablet and then the emotive reactions after analyzing the user answers. Three different plans are produced, one for each type of user level, the contingent planning is exploited to manipulate the robot's emotions according to the user's level and answers received. The types and predicates of this domain are shown in the following snippet.

```
(:types
question
)

(:predicates
    (right_answer ?x - question)      ; true if the user chooses the right answer to question x
    (question_answered ?x - question)      ; true if the user answered the question
    (difficulty_easy)      ; true if the quiz difficulty is set to easy
    (difficulty_medium)      ; true if the quiz difficulty is set to medium
```

```
    (difficulty_hard)      ; true if the quiz difficulty is set to hard
    (quiz_started)
    (quiz_ended)      ; true if the quiz has ended

    (classification_done)      ; true if the interview to know the person has been completed

)
```

Here follows a brief table with descriptions of all the actions of the pepper domain.

Table 4.2: Description of the pepper_domain actions.

Action	Description
quiz_start	Start the quiz
show_question	Show the given question to the user
check_answer	Check if the given answer is correct
sad_emotional_response_easy	Perform a sad emotional response if the quiz was easy
happy_emotional_response_easy	Perform a happy emotional response if the quiz was easy
sad_emotional_response_medium	Perform a sad emotional response if the quiz was medium
happy_emotional_response_medium	Perform a happy emotional response if the quiz was medium
sad_emotional_response_hard	Perform a sad emotional response if the quiz was hard
happy_emotional_response_hard	Perform a happy emotional response if the quiz was hard

The `planner_for_server` module is the python planner for this phase. It works along with the `app` module to receive the user's input data from the HTML pages and to process them to take the appropriate action in response. Every web page, but the `quiz_end.html`, share the same function **`sendDataToServer`**, which is called by the `EventListener` linked to the button click.

```
function sendDataToServer(data, nextPage) {
        fetch('/endpoint', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ 'data': data }),
        })
        .then(response => response.json())
        .then(data => {
            console.log('Success:', data);
```

```
12              // Change the page after sending data to the server
13              if (nextPage) {
14                  window.location.href = nextPage;
15              }
16          })
17          .catch((error) => {
18              console.error('Error:', error);
19          });
20      }
```

In this way the web pages can communicate with the `app` module, which in turn communicates with `planner_for_server`; `app` receives data through the `handle_request` function, that saves them to process them at the end and calls the `next_action` function from `planner_for_server` to ask directly to the planner, reading the PDDL plan, what should be the next action.

After that the quiz it proceeds executing the `show_question` action, rendering templates using Flask and receiving and processing data through the process explained before, using the `check_answer` actions. At the end of the quiz the planner computes the final score of the user and, depending on the quiz difficulty, takes a sad or a happy reaction.

Chapter 5

Results

In this section, we will highlight some of the aspects of the Human-Pepper interaction run on the Choregraphe environment. A demo is available at this link.

5.1 Human Robot Interaction

5.1.1 Detecting User

As we already said in 3 and in 4, initially the robot search its surroundings for potential users using its sonars through the `sense` function, users that are detected once they approach Pepper in a distance range of 1 meter. During the scanning period, Pepper waits in silence, and all processes are communicated just in the terminal window 5.1.

```
robot@luca-VirtualBox:~/playground$ python main.py
begin
Connecting to robot 127.0.0.1:9559 ...
Connected to robot 127.0.0.1:9559
Starting services...
Waiting for signal...
Person approaching...
They was just passing by.
Waiting for signal...
```

Figure 5.1: Screenshot showing the terminal during the use of the Sensor class.

Once the user approaches the robot, there are two main scenarios: the user is behind or in front

of Pepper, and here we show screenshots of the simulator for both cases.

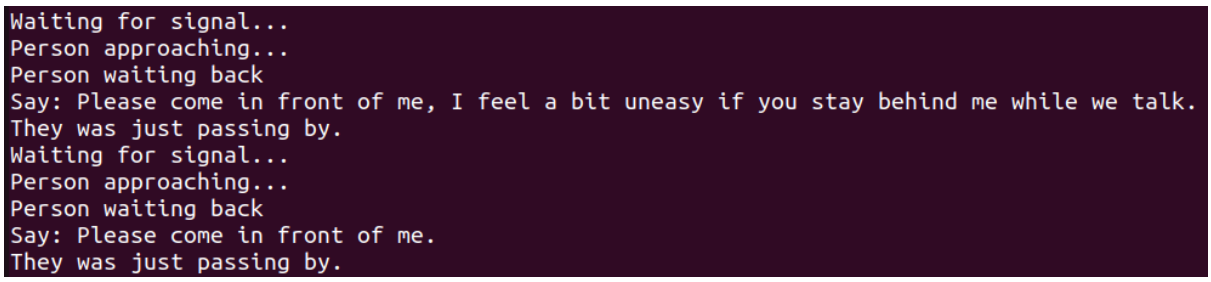

Figure 5.2: Screenshot showing the simulated person approach from behind

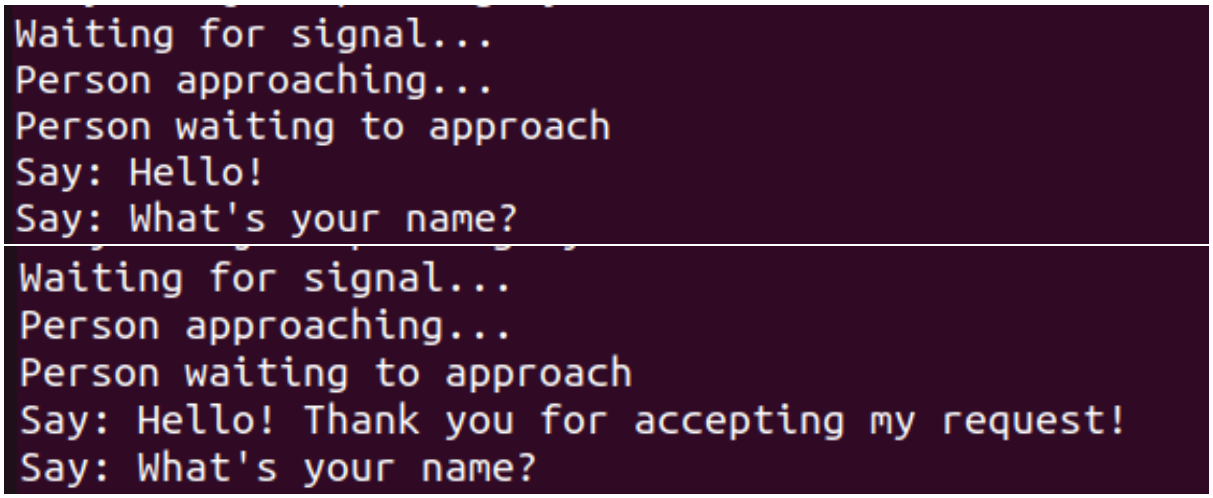

Figure 5.3: Screenshot showing the simulated person approaches from the front, the first in the usual case, the second if the user approached the robot from behind before

5.1.2 Pepper change of behaviour

Again, Pepper can react differently based on the user's response and course of interaction. For example, during the *Person Approaching* phase, if the user took a quiz already, Pepper can react as shown in Figure 5.4. If the user made a low score in the quiz, Pepper acts coldly, and it will change its behavior if and only if the user will make an higher score. Otherwise, if the user took a high score, Pepper acts more politely, but still it can change its behavior if the user will take a lower score.

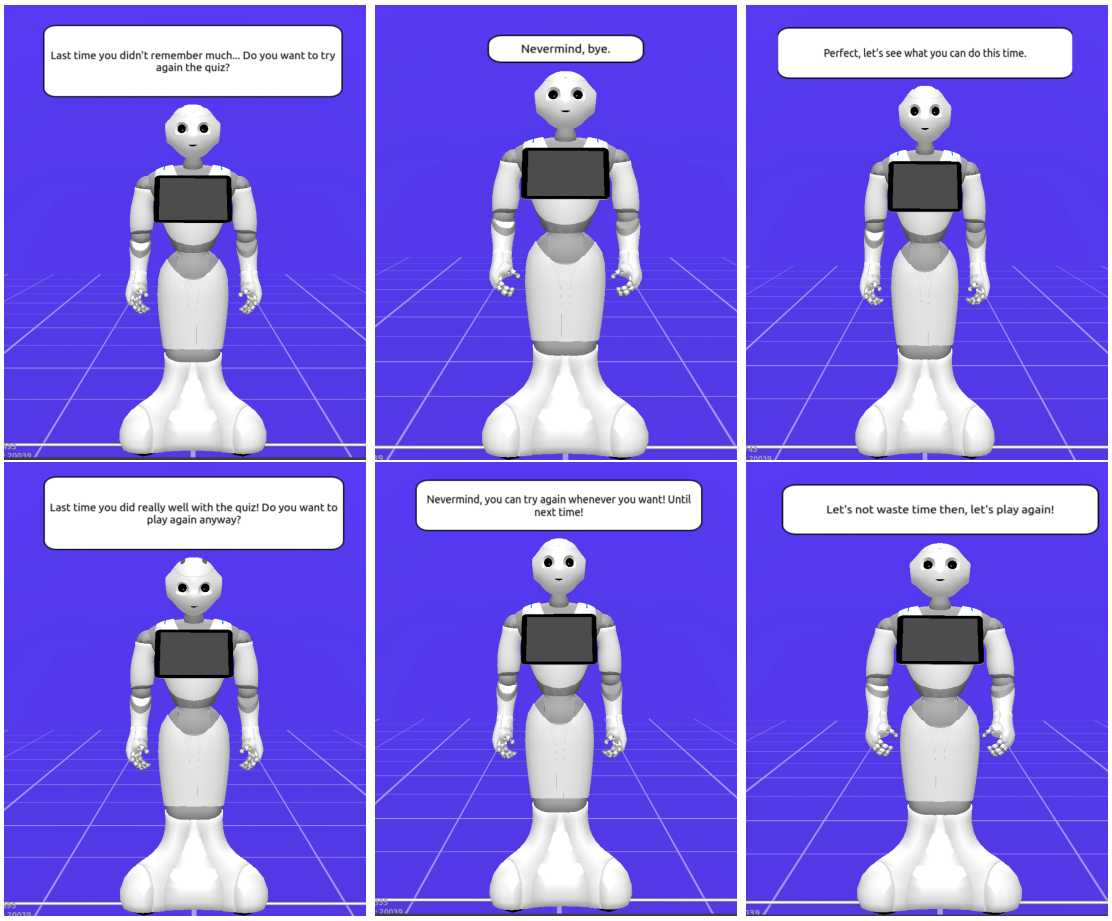

Figure 5.4: 1st column: user approaches Pepper again after having completed the quiz; 2nd column: user do not want to take the quiz again; 3rd column: user accepts to repeat the quiz

In addition to the change in the mood and its way of expressing itself through natural dialogue, Pepper expresses its emotion in a more flashy/direct manner, that is, through emotional reaction. We can see its three kind of responses: *neutral* 5.5, in which it does not move at all, but simply says "Understood" and waits a few seconds; *positive* 5.6, in which it makes a sequence of two movements to express joy and happiness, but what it says can vary depending on the situation; and finally *negative* 5.5, in which it changes its posture by crouching, assuming a position that makes its head facing down, accompanied by its arms, miming an emotional reaction that expresses discouragement and sadness, and as for the *positive* case, it says different things according to the user's action.

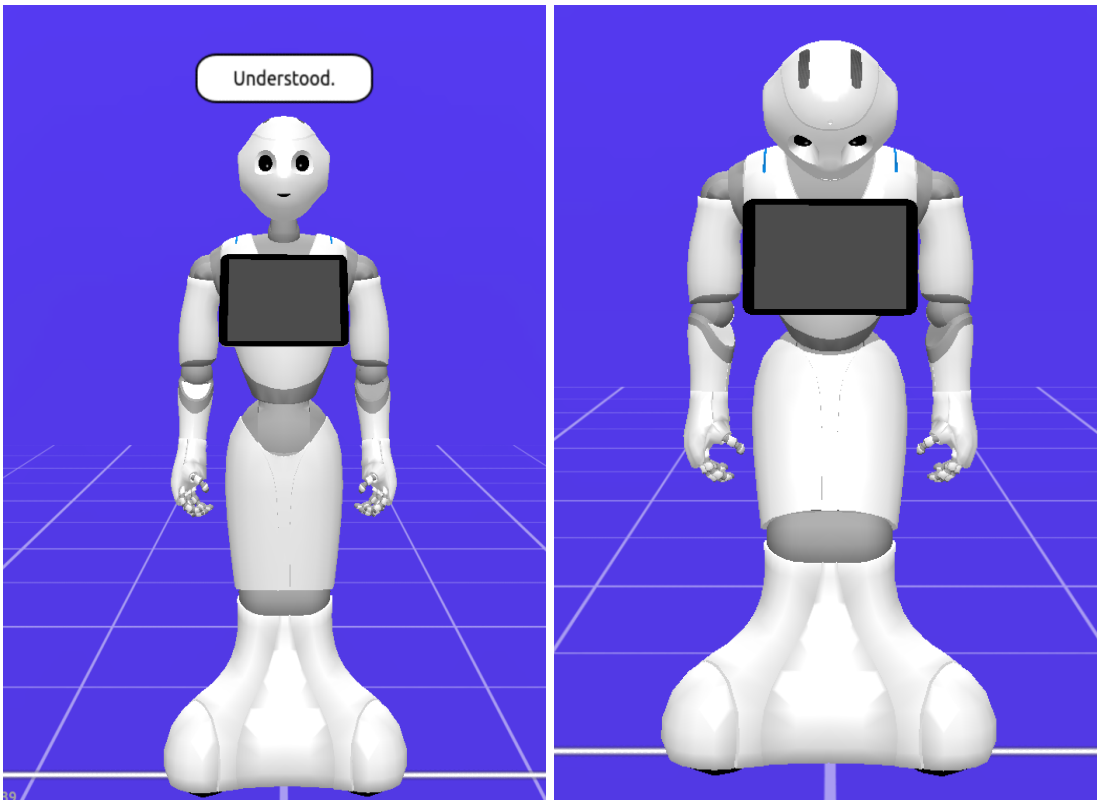

Figure 5.5: Simulator showing Pepper's neutral emotive reaction on the left, and the sad one on the right

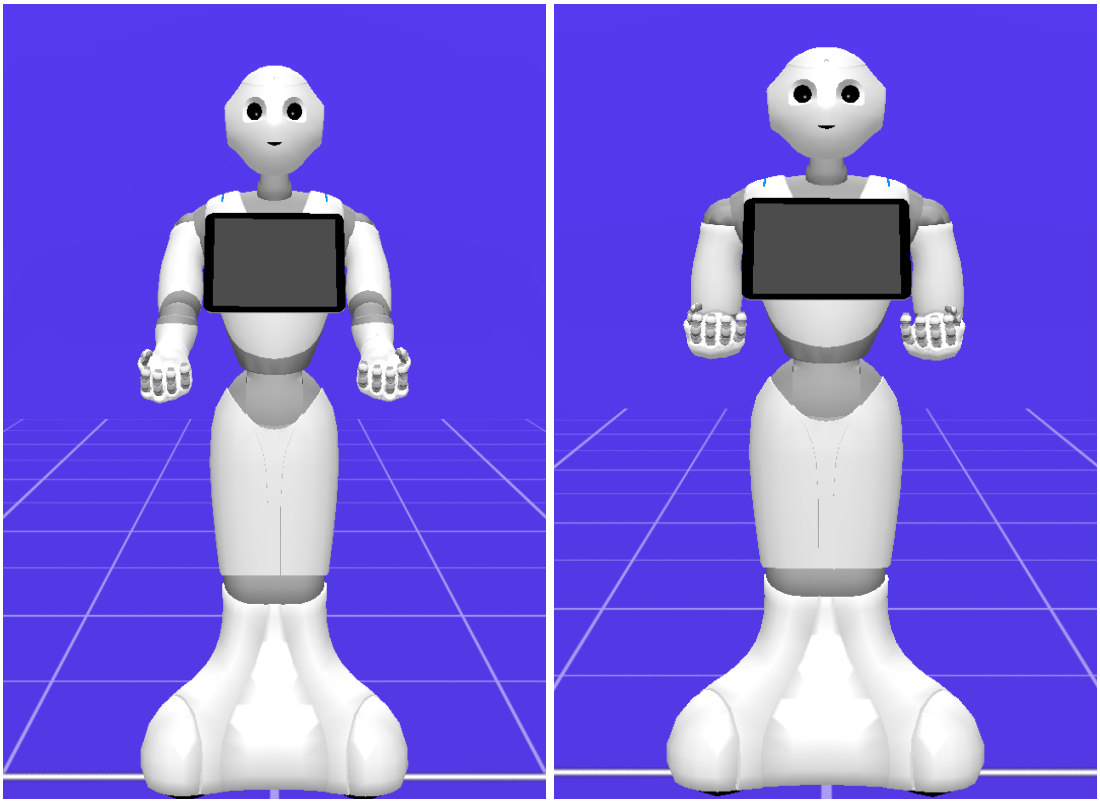

Figure 5.6: Simulator showing Pepper's happy emotive reaction

5.1.3 Quiz and Final response

Once the quiz the second phase reaches its end, Pepper computes the user level and proposes a quiz accordingly. The three quizzes have increasing difficulty, which means that not only the user must take an higher score to obtain a happy reaction from the robot but also that the questions are increasingly harder. In Figure 5.7 we show a comparison between a question on one of the fun fact provided by the robot of different difficulty.

As we said in 4, the HTML question pages share all the same style, while the other pages communicating some kind of information to the user, such as the start/end of the quiz or the right/wrong answer, have another style in common. We show examples of both styles in Figures 5.7 and 5.8.

Figure 5.7: multiple-choices question web pages in comparison, from the left: easy, medium and hard.

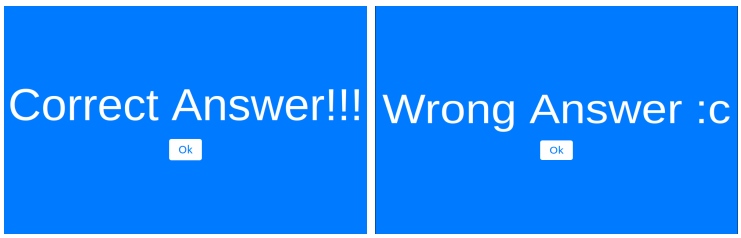

Figure 5.8: Firefox browser simulating Pepper's tablet appearance on informational web pages

At the end of the quiz, the robot reacts to the user final score, and its behaviour reflects the user's level of appreciation of robots along with the user's attention, as we can see from Figure 5.9.

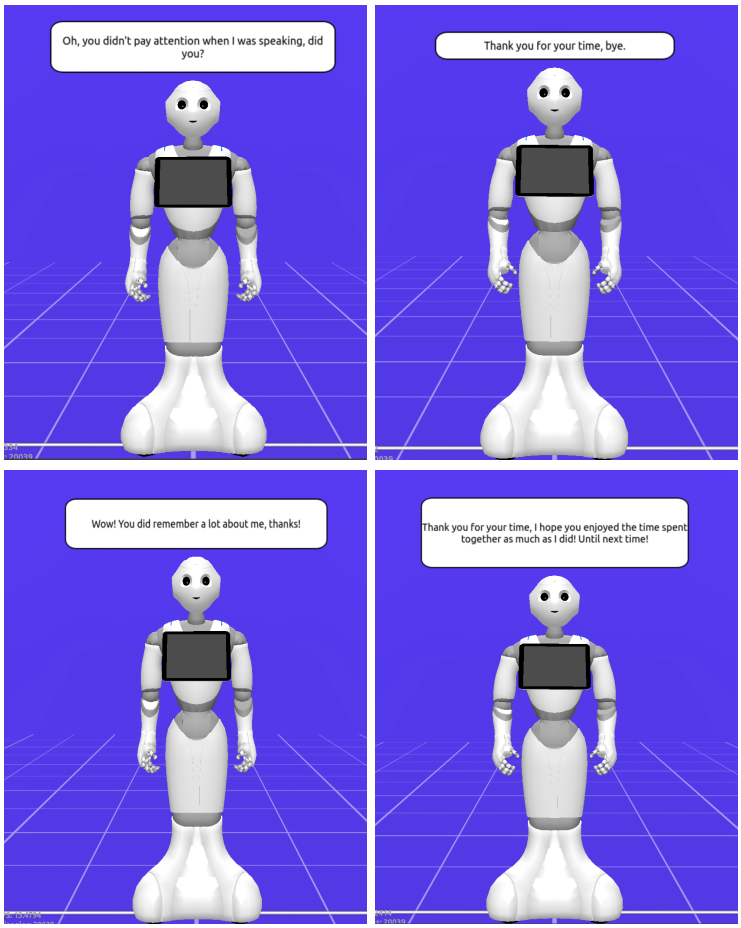

Figure 5.9: Simulator showing Pepper's final farewell, on top if the quiz made Pepper sad, on the bottom if it made it happy.

5.2 Reasoning Agents

The PDDL planners worked as expected, leading the robot to correctly classify the user level and then, to compute its score in all the three different quizzes.

5.2.1 Interview planner

As we have seen in 4, the `interview_planner` module handles both the start of the interview and its end. We can see it working in the terminal in Figures 5.10 and 5.11

```
Starting services...
('id: ', '0||0', 'action: ', 'INTERVIEW_USER INT', 'son: ', 'SON: 1||0')
('executing action: ', 'INTERVIEW_USER')
```

Figure 5.10: Terminal showing the execution of the PDDL plans, used to start the interview.

```
### Interview ended ###
('id: ', '1||0', 'action: ', 'CHECK_USER_INTERVIEW_HARD INT')
('trueson: ', 'TRUESON: 2||0', 'falseson: ', 'FALSESON: 2||1')
('executing action: ', 'CHECK_USER_INTERVIEW_HARD')
('id: ', '2||1', 'action: ', 'CLASSIFY_HARD_FALSE INT', 'son: ', 'SON: 3||0')
('executing action: ', 'CLASSIFY_HARD_FALSE')
('id: ', '3||0', 'action: ', 'CHECK_USER_INTERVIEW_MEDIUM INT')
('trueson: ', 'TRUESON: 4||0', 'falseson: ', 'FALSESON: 4||1')
('executing action: ', 'CHECK_USER_INTERVIEW_MEDIUM')
('id: ', '4||1', 'action: ', 'CLASSIFY_MEDIUM_FALSE INT', 'son: ', 'SON: 5||0')
('executing action: ', 'CLASSIFY_MEDIUM_FALSE')
('id: ', '5||0', 'action: ', 'CLASSIFY_EASY_TRUE INT', 'son: ', 'SON: 6||-1')
('executing action: ', 'CLASSIFY_EASY_TRUE')
('The user was classified as: ', 'easy')
```

Figure 5.11: Terminal showing the execution of the PDDL plans, used to compute the user level.

5.2.2 Quiz planner

Here the planner attends each step, controlling every aspect of the tablet interaction: it chooses the next question, through the *show_question* action, checks every answer, through the *check_answer* action, and finally it performs the last emotional reaction through one of the *emotional_response* actions. We can see it in Figures 5.12, 5.13 and 5.14.

```
('Data received from client:', u'Quiz start')
### Last executed action: 0||0 ###
('id: ', '0||0', 'action: ', 'QUIZ_START', 'son: ', 'SON: 1||0')
('id: ', '1||0', 'action: ', 'SHOW_QUESTION QUESTION_GUINNESS', 'son: ', 'SON: 2||0')
('### Action_to_execute: ', 'SHOW_QUESTION QUESTION_GUINNESS', '###')
127.0.0.1 - - [21/Dec/2023 14:06:03] "POST /endpoint HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2023 14:06:03] "GET /quiz HTTP/1.1" 200 -
('Data received from client:', u'Correct answer')
127.0.0.1 - - [21/Dec/2023 14:06:34] "POST /endpoint HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2023 14:06:34] "GET /correct_answer HTTP/1.1" 200 -
('Data received from client:', u'True_son')
### Last executed action: 1||0 ###
('id: ', '0||0', 'action: ', 'QUIZ_START', 'son: ', 'SON: 1||0')
('id: ', '1||0', 'action: ', 'SHOW_QUESTION QUESTION_GUINNESS', 'son: ', 'SON: 2||0')
('id: ', '2||0', 'action: ', 'SHOW_QUESTION QUESTION_MOON', 'son: ', 'SON: 3||0')
('### Action_to_execute: ', 'SHOW_QUESTION QUESTION_MOON', '###')
127.0.0.1 - - [21/Dec/2023 14:06:35] "POST /endpoint HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2023 14:06:35] "GET /quiz HTTP/1.1" 200 -
('Data received from client:', u'Wrong answer')
127.0.0.1 - - [21/Dec/2023 14:06:38] "POST /endpoint HTTP/1.1" 200 -
127.0.0.1 - - [21/Dec/2023 14:06:38] "GET /wrong_answer HTTP/1.1" 200 -
('Data received from client:', u'False_son')
### Last executed action: 2||0 ###
('id: ', '1||0', 'action: ', 'SHOW_QUESTION QUESTION_GUINNESS', 'son: ', 'SON: 2||0')
('id: ', '2||0', 'action: ', 'SHOW_QUESTION QUESTION_MOON', 'son: ', 'SON: 3||0')
('id: ', '3||0', 'action: ', 'SHOW_QUESTION QUESTION_MADE', 'son: ', 'SON: 4||0')
```

Figure 5.12: Terminal showing the execution of the PDDL plans showing the quiz's questions.

```
### Executing action: CHECK_ANSWER QUESTION_GUINNESS ###
### Answer is TRUE ###
### Last executed action: 5||0 ###
('id: ', '4||0', 'action: ', 'SHOW_QUESTION QUESTION_NAME', 'son: ', 'SON: 5||0')
('id: ', '5||0', 'action: ', 'CHECK_ANSWER QUESTION_GUINNESS')
('trueson: ', 'TRUESON: 6||0', 'falseson: ', 'FALSESON: 6||1')
('id: ', '6||0', 'action: ', 'CHECK_ANSWER QUESTION_MOON')
('trueson: ', 'TRUESON: 7||0', 'falseson: ', 'FALSESON: 7||1')
('### Action_to_execute: ', 'CHECK_ANSWER QUESTION_MOON', '###')
### Executing action: CHECK_ANSWER QUESTION_MOON ###
### Answer is FALSE ###
### Last executed action: 6||0 ###
('id: ', '5||0', 'action: ', 'CHECK_ANSWER QUESTION_GUINNESS')
('trueson: ', 'TRUESON: 6||0', 'falseson: ', 'FALSESON: 6||1')
('id: ', '6||0', 'action: ', 'CHECK_ANSWER QUESTION_MOON')
('trueson: ', 'TRUESON: 7||0', 'falseson: ', 'FALSESON: 7||1')
('id: ', '7||1', 'action: ', 'CHECK_ANSWER QUESTION_MADE')
('trueson: ', 'TRUESON: 8||0', 'falseson: ', 'FALSESON: 8||1')
```

Figure 5.13: Terminal showing the execution of the PDDL plans checking the answers.

```
### Last executed action: 8||1 ###
('id: ', '7||1', 'action: ', 'CHECK_ANSWER QUESTION_MADE')
('trueson: ', 'TRUESON: 8||0', 'falseson: ', 'FALSESON: 8||1')
('id: ', '8||1', 'action: ', 'CHECK_ANSWER QUESTION_NAME')
('trueson: ', 'TRUESON: 9||0', 'falseson: ', 'FALSESON: 9||1')
('id: ', '9||0', 'action: ', 'HAPPY_EMOTIONAL_RESPONSE_EASY QUESTION_NAME QUESTION_GUINNESS QUESTION_MOON QUESTION_MADE', 'son: ', 'SON: 10||-1')
('### Action_to_execute: ', 'HAPPY_EMOTIONAL_RESPONSE_EASY QUESTION_NAME QUESTION_GUINNESS QUESTION_MOON QUESTION_MADE', '###')
### Executing action: HAPPY_EMOTIONAL_RESPONSE_EASY QUESTION_NAME QUESTION_GUINNESS QUESTION_MOON QUESTION_MADE ###
1
Positive emotive reaction
Say: Wow! You did remember a lot about me, thanks!
Posture changed in joints: ['LWristYaw', 'RWristYaw', 'LShoulderPitch', 'RShoulderPitch']
Posture changed in joints: ['LWristYaw', 'RWristYaw', 'LShoulderPitch', 'RShoulderPitch', 'LElbowRoll', 'RElbowRoll']
Say: Thank you for your time, I hope you enjoyed the time spent together as much as I did! Until next time!
Posture changed in joints: ['LWristYaw', 'RWristYaw', 'LShoulderPitch', 'RShoulderPitch']
Posture changed in joints: ['LWristYaw', 'RWristYaw', 'LShoulderPitch', 'RShoulderPitch', 'LElbowRoll', 'RElbowRoll']
```

Figure 5.14: Terminal showing the execution of the PDDL plans executing the final emotional reaction.

Chapter 6

Conclusions

The integration of social robots into our daily lives has the potential to revolutionize human-machine interactions. Our work, though small, with a focus on the implementation and interaction design of a social robot capable of having an emotional connection with the user has demonstrated that:

- A robot with human-like features helps people resonate better with it.
- A robot showing human emotions, even if not real, can establish meaningful connections with humans.
- Some kind of planner is necessary to handle real-time user interactions.

This type of interaction is very promising, it can be exploited, for example, to make robots adapt to the users' routing and personal characteristics to collect personal information without using their trust.

In future works, utilizing a real robot would certainly better prove our points, since we made all of our experiments only on a simulated environment. Our work could also be expanded by exploiting the latest large language generative models so that dialogue could be more interactive and realistic.

References

Eroglu, Sevgin A., Machleit, Karen A., and Davis, Lenita M. (2001). "Atmospheric qualities of online retailing: A conceptual model and implications". In: *Journal of Business Research* 54.2. Retail Consumer Decision Processes, pp. 177–184. ISSN: 0148-2963. DOI: https://doi.org/10.1016/S0148-2963(99)00087-9. URL: https://www.sciencedirect.com/science/article/pii/S0148296399000879.

Fong, Terrence, Thorpe, Charles, and Baur, Charles (2003). "Collaboration, Dialogue, Human-Robot Interaction". In: *Robotics Research*. Ed. by Raymond Austin Jarvis and Alexander Zelinsky. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 255–266. ISBN: 978-3-540-36460-3.

Ghani, Dahlan and Ishak, Sidin (June 2012). "Relationship Between The Art of Wayang Kulit and Disney's Twelve Principles of Animation". In: *REVISTA DE CERCETARE SI INTERVENTIE SOCIALA* 37, pp. 162–179.

Gockley, Rachel, Simmons, Reid, and Forlizzi, Jodi (2006). "Modeling Affect in Socially Interactive Robots". In: *ROMAN 2006 - The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pp. 558–563. DOI: 10.1109/ROMAN.2006.314448.

Grinberg, Miguel (2018). *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.".

Hoffmann, Jörg and Brafman, Ronen I. (2006). "Conformant planning via heuristic forward search: A new approach". In: *Artificial Intelligence* 170.6, pp. 507–541. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2006.01.003. URL: https://www.sciencedirect.com/science/article/pii/S0004370206000087.

Kirby, Rachel, Forlizzi, Jodi, and Simmons, Reid (2010). "Affective social robots". In: *Robotics and Autonomous Systems* 58.3. Towards Autonomous Robotic Systems 2009: Intelligent, Autonomous Robotics in the UK, pp. 322–332. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2009.09.015. URL: https://www.sciencedirect.com/science/article/pii/S0921889009001547.

Kraus, Matthias, Dettenhofer, Viktoria, and Minker, Wolfgang (2022). "Responsible Interactive Personalisation for Human-Robot Cooperation". In: UMAP '22 Adjunct. Barcelona, Spain: Association for Computing Machinery, pp. 58–62. ISBN: 9781450392327. DOI: 10.1145/3511047.3536419. URL: https://doi.org/10.1145/3511047.3536419.

Majercik, Stephen M. and Littman, Michael L. (2003). "Contingent planning under uncertainty via stochastic satisfiability". In: *Artificial Intelligence* 147.1. Planning with Uncertainty and Incomplete Information, pp. 119–162. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(02)00379-X. URL: https://www.sciencedirect.com/science/article/pii/S000437020200379X.

Matsui, D., Minato, Takashi, MacDorman, Karl, and Ishiguro, H. (Sept. 2005). "Generating natural motion in an android by mapping human motion". In: pp. 3301–3308. DOI: 10.1109/IROS.2005.1545125.

Pot, E., Monceaux, J., Gelin, R., and Maisonnier, B. (2009). "Choregraphe: a graphical tool for humanoid robot programming". In: *RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication*, pp. 46–51. DOI: 10.1109/ROMAN.2009.5326209.

Stock-Homburg, Ruth (Mar. 2022). "Survey of Emotions in Human–Robot Interactions: Perspectives from Robotic Psychology on 20 Years of Research". In: *International Journal of Social Robotics* 14.2, pp. 389–411. ISSN: 1875-4805. DOI: 10.1007/s12369-021-00778-6. URL: https://doi.org/10.1007/s12369-021-00778-6.

Yoon, Youngwoo, Ko, Woo-Ri, Jang, Minsu, Lee, Jaeyeon, Kim, Jaehong, and Lee, Geehyuk (2018). *Robots Learn Social Skills: End-to-End Learning of Co-Speech Gesture Generation for Humanoid Robots*. arXiv: 1810.12541 [cs.RO].