Group E – Mamoru Ota, Mohamed ElShab, Bruno Huntzinger

First Assignment

I - The way different design patterns could play a role in the final architecture implementation:

1) The singleton pattern:

The singleton pattern is used when a class must have only one instance in a whole system. It is rather easy to find a relevant usage of this pattern in the case of a sushi restaurant where the delivery is assured by autonomous robots. In our case, all the TIAGo robot are managed by a unique orchestration manager (although the component diagram shows only one TIAGo robot for the sake of simplicity…). Therefore, we have to ensure that the Python class describing the Orchestration Manager can have only one instance. Our code implements the singleton pattern to address this issue. By doing this, we ensure that there can not be two instances of the Orchestration Manager class giving orders at the same time, disrupting the proper functioning of the restaurant.

In the same way, each TIAGo robot has a unique Manipulation System to control its actuators, a unique Navigation System to manage the trajectory planning, a unique Order Verification System to verify that the orders have been correctly executed, a unique system to reason about the food placement. We have therefore implemented the singleton pattern for each of these system.

2) The adapter pattern:

The adapter pattern is used as a sort of software bridge that allows objects with incompatible interfaces to work and interact together. We can find several place to use it in our architecture.

Firstly, the adapter pattern can be potentially used in the Manipulation System to transform the data coming from the force sensors and the sensors integrated to the joint actuators into data that the other components of the system can understand.

Secondly, it can also be integrated - with a higher level of complexity - into the Speech Interface, for example, to transform the recordings of what the clients say into words so that an analysis component can determines the meaning of what the clients tell

the robots.

3) Mediator Pattern:

In our software architecture, the Orchestration Manager plays the role of the Mediator between the point of sale and the TIAGo robots by managing the communication beetween them. It boosts the efficiency of the whole system by assigning to each robot its task (every robot doesn't have to determine if it is able to perform a task and if it is the best placed to perform the task), significantly decreasing the number of calculation that need to be performed by the whole system. It also enhances the scalability because adding a new TIAGo platform or a new point of sale amounts just to add a new ROS node.

4) Observer Pattern:

In our software architecture, the Observer pattern is widely used, in the form of communication between modules via ROS. This kind of communication allows a greater flexibility in term of execution, suppressing the need for synchronization between the point of sale publishing frequency and the Orchestration Manager reading frequency, and between the Orchestration Manager publishing frequency and the reading frequency of the TIAGo robots.

II - List of the components described according to the component-based software architecture paradigm:

1) PointOfSale (POS):

- **Provided interface:**
    - /orders (ROS topic): publishes new customer orders.
        - **Type:** data interface (publishes order data).
        - **State:** stateless (each published order is independent).
        - **Typing:** loosely-typed (uses std_msgs/String but contains structured data within the string: "Table : X, dish : Y, ..."). Requires parsing by the subscriber.

2) OrchestrationManager:

- **Required interfaces:**
    - /orders (ROS topic): subscribes to receive new orders from the POS. (See *POS* description above).
    - /availability (ROS topic): subscribes to receive robot availability status.
        - **Type:** data interface (receives robot state data).
        - **State:** stateless (each message represents current status).
        - **Typing:** loosely-typed (receives std_msgs/String type message like "TIAGo X : available").
    - /position (ROS topic): subscribes to receive robot positions.
        - **Type:** data interface (receives robot state data).
        - **State:** stateless (each message represents the current position).
        - **Typing:** strongly-typed (uses geometry_msgs/Point).
- **Provided interfaces:**
    - /order_TIAGo (ROS topic): publishes specific task assignments to robots.
        - **Type:** data interface (publishes order messages). Could also be seen as triggering a service, but by using a topic.
        - **State:** stateless (each message is a new assignment).
        - **Typing:** loosely-typed (uses std_msgs/String like "TIAGo n°X, table : Y, dish : Z").
    - /error_messages (ROS topic): publishes error messages for staff/logging.
        - **Type:** data interface.
        - **State:** stateless (each message is an independent error report).
        - **Typing:** loosely-typed (uses std_msgs/String, but typically carries unstructured text).

3) TIAGo (Main Robot):

- **Required interfaces:**
    - /order_TIAGo (ROS topic): subscribes to receive task assignments from OrchestrationManager. (See *OrchestrationManager* description).

- **Provided interfaces:**
    - /availability (ROS topic): publishes its current status ("available" or "occupied").
    - /position (ROS topic): publishes its current position (x, y) and ID (in z), see *OrchestrationManager* description.
    - /orders (ROS topic): publishes requests for table clearing.
        - **Type:** data interface.
        - **State:** stateless.
        - **Typing:** strongly-typed (uses geometry_msgs/Point).
- **Internal interfaces:**
    - Calls methods on PerceptionSystem, OrderVerificationSystem, NavigationSystem, ManipulationSystem, ReasoningSystem with stateful and strongly-typed service interfaces.

4) PerceptionSystem:

- **Provided interfaces (conceptual interfaces – often created via the method calls from TIAGo):**
    - perception_for_navigation(): returns an obstacle detection boolean.
        - **Type:** service interface (provides a computation result on demand).
        - **State:** stateless (result depends only on current sensor simulation).
        - **Typing:** strongly-typed (returns a boolean).
    - verification_of_grasping_and_placement(operation): returns a success boolean.
        - **Type:** service interface.
        - **State:** stateless (result based on probability).
        - **Typing:** strongly-typed (takes as input a string, returns a boolean).

5) NavigationSystem:

- **Provided interfaces:**
    - /mp (ROS topic): Publishes the nav_msgs/OccupancyGrid.
        - **Type:** data interface.

        - **State:** stateful (the map represents accumulated knowledge).
        - **Typing:** strongly-typed (nav_msgs/OccupancyGrid).
    - /robot_pose (ROS topic): publishes the estimated pose via a geometry_msgs/PoseStamped message.
        - **Type:** data interface.
        - **State:** stateful (the pose depends on the previous states of the TIAGo robots).
        - **Typing:** strongly-typed (geometry_msgs/PoseStamped).
    - /trajectory (ROS topic): publishes the planned nav_msgs/Path.
        - **Type:** data interface.
        - **State:** stateless (represents the current plan for the current goal).
        - **Typing:** strongly-typed (nav_msgs/Path).
- **Required interfaces:**
    - /goal (ROS topic): subscribes to receive navigation goals (geometry_msgs/PoseStamped).
        - **Type:** data interface (receives command data).
        - **State:** Stateless (each goal is independent).
        - **Typing:** Strongly-typed (geometry_msgs/PoseStamped).
    - Sensor topics (e.g., /imu_data, /left_encoder, /right_encoder, /object_detection, /manipulation_status - based on SLAM subscribers in the code): these are stateless data interfaces and their strongly/loosely typed nature depends on the sensor considered.
- **Provided interfaces (to the TIAGo program via methods):**
    - navigate_to(): strongly-typed service interface which triggers stateful navigation behavior,
    - update_map(), update_position(): stateless and strongly-typed service interfaces.
    - stop(): service interface, stateless trigger, strongly-typed.

6) ManipulationSystem:

- **Provided interfaces:**

  - /force_data (ROS topic): publishes std_msgs/Float32, using stateless and strongly-typed data.
  - /motors_feedback (ROS topic): publishes std_msgs/String messages which contains structured stateful (it reflects motor state) using loosely-typed data.
  - /safety_status (ROS topic): publishes std_msgs/String messages which contains stateful and strongly-typed data in the form of an enumeration-like string.
  - /manipulation_status (ROS topic): publishes std_msgs/String messages which contains stateful and strongly-typed data in the form of an enumeration-like string.

- **Required interfaces:**
  - /gripper_commands (ROS topic): subscribes to std_msgs/Float32 messages which contains stateless, strongly-typed data/commands.
  - /grasp_commands (ROS topic): subscribes to std_msgs/Float32 messages which contains stateless, strongly-typed data/commands.
  - /joint_commands (ROS topic): subscribes to std_msgs/String messages which contains stateless, loosely-typed data/commands (strings that encode positions).
  - /trajectory_commands (ROS topic): subscribes to std_msgs/String messages which contains stateless, loosely-typed data/commands (string that encodes waypoints).
  - /target_dish_position (ROS topic): subscribes to geometry_msgs/Point messages which contains stateless, strongly-typed data.
  - /force_data (ROS topic): internal subscription (see above).
  - /safety_status (ROS topic): internal subscription (see above).
  - /motors_feedback (ROS topic): internal subscription (see above).
  - /perception_data (ROS topic): subscribes to std_msgs/String messages which contains stateless, loosely-typed data (generic placeholder).

- **Provided Interfaces (to TIAGo program via methods):**
  - execute_manipulation(): strongly-typed service interface, which triggers a stateful manipulation.

7) ReasoningSystem:

- **Provided interface:**
    - /target_dish_position (ROS topic): publishes geometry_msgs/Point messages which contains stateless (result of a specific request) and strongly-typed data
    - reason_about_placement(): stateless (performs calculation based on input) and strongly typed service interface, provided to the TIAGo program via methods.
- **Required Interfaces:** Non explicitly shown via ROS (depends on how perception_data is passed to the reason_about_placement method).

8) OrderVerificationSystem:

- **Required interfaces:**
    - access to TIAGo instance attributes (perception_system, id, target_table) via an internal interface or dependency.
    - interaction with SpeechInterface (via method call verify_delivery_client) using a stateful and strongly-typed service interface.
    - interaction with PerceptionSystem (via method call like verify_item_served) using a stateless and strongly-typed service interface.
- **Provided interfaces (to TIAGo via methods):**
    - verify_served_order(): stateless and strongly-typed service interface.
    - verify_delivery_client(): stateless and strongly-typed service interface.
- **Provided interface:**
    - /error_messages (ROS topic): publishes a std_msgs/String message which contains stateless and loosely-typed data.

9) SpeechInterface:

- **Provided interfaces (to the OrderVerificationSystem):**
    - verify_delivery_client(): stateful and strongly-typed service interface.

III - Main key performance indicators (KPIs) of the three main components:

The criteria that will be evaluated for these components will be the correctness, the functionality, the security and robustness.

▪ Orchestration Manager:

▪ Orders handling:

- Percentage of order sent by the points of sale successfully received by the Orchestration Manager: computed by taking the ratio between the number of orders received by the Orchestration Manager and the number of orders sent by the points of sale and by multiplying the total by 100. By computing this parameters, one can verify if the Orchestration Manager successfully receives every order messages published by the points of sales.

- Percentage of incorrect orders received and successfully analyzed: computed by taking the ratio between the number of incorrect orders correctly processed and the number of incorrect orders received and by multiplying the total by 100. By sending deliberately wrong orders and computing this parameter, one can effectively assess the robustness of the Orchestration Manager's order handling is case of unexpected errors in the order sent by the point of sales.

- Percentage of correct orders correctly analyzed: computed by taking the ratio between the number of correct orders correctly processed and the number of correct orders received and by multiplying the total by 100. By computing this parameter, one can assess how well the Orchestration Manager analyzes correct orders and proceed in nominal conditions.

▪ Handling of the availability updates of the TIAGo robots:

- Percentage of availability messages sent by the TIAGo robots successfully received by the Orchestration Manager: computed by taking the ratio between the number of availability messages received by the Orchestration Manager and the number of availability message sent by the TIAGo robots and by multiplying the total by 100. By computing this parameters, one can verify if the Orchestration Manager successfully receives every availability messages published by the TIAGo robots.

- Percentage of correct availability messages successfully analyzed: computed by taking the ratio between the number of correct availability messages successfully analyzed by the Orchestration Manager and the number of correct availability messages received by the Orchestration Manager and by multiplying the total by 100. By computing this number, one can verify if the statuses of the TIAGo robots are correctly processed in the nominal conditions.

- Percentage of incorrect availability messages generating the publication of an error message: computed by taking the ratio between the number of incorrect availability messages generating the publication of an error message and the number of incorrect availability messages received by the Orchestration Manager and by multiplying the total by 100. By computing this number, one can verify if the Orchestration efficiently raises error messages in the case of a TIAGo robot updating him with a wrong status.

▪ Handling of the position updates of the TIAGo robots:

- Percentage of position messages sent by the TIAGo robots successfully received by the Orchestration Manager: computed by taking the ratio between the number of position messages received by the Orchestration Manager and the number of position messages sent by the TIAGo robots and by multiplying the

total by 100. By computing this parameters, one can verify if the Orchestration Manager successfully receives every position messages published by the TIAGo robots.

- Percentage of correct position messages successfully analyzed: computed by taking the ratio between the number of correct position messages successfully analyzed by the Orchestration Manager and the number of correct position messages received by the Orchestration Manager and by multiplying the total by 100. By correct position messages, we mean that the position tuple received does not give the robot a position where some parts of it intersects with a wall or a table. By computing this number, one can verify if the positions of the TIAGo robots are correctly processed and updated in the nominal conditions.

- Percentage of incorrect position messages generating the publication of an error message: computed by taking the ratio between the number of incorrect position messages generating the publication of an error message and the number of incorrect position messages received by the Orchestration Manager and by multiplying the total by 100. By computing this number, one can verify if the Orchestration efficiently raises error messages in the case of a TIAGo robot updating him with a position which make it intersect with a wall or a table, both these situation being physically impossible.

▪ Distance computation:

- Percentage of distances computed correctly: computed by taking the ratio of the number of correctly computed distances over the number of distances computed, the total multiplied by 100. By computing this parameter, one can verify if the Orchestration Manager computes successfully the distance between two objects. This ability is essential for giving the orders to optimal TIAGo robot.

▪ Publication of orders to the TIAGo robots:

- Idleness coefficient: computed by taking the minimum between the number of TIAGo robots available at the end of a loop and the number of orders that remains to be given to a TIAGo robot. The lower this parameter, the better the order publication ability of the Orchestration Manager. If this parameter is greater than zero, it means that it remains at least one TIAGo robot available and at least one order that has to be given at the end of the control loop, meaning that the Orchestration Manager is not performing correctly.

▪ Analysis of the message transmission part of the component:

-Percentage of situations where the messages containing the orders for the TIAGo robots have been correctly sent: computed by dividing the number of times the order messages have been correctly sent by the numbers of attempts to send the order messages and multiplying the total by 100. By computing this parameter, one may assess the ability of the Orchestration Manager to send the orders to the TIAGo robots.

-Percentage of situations where the error messages have been correctly sent: computed by dividing the number of times the error messages have been correctly sent by the numbers of attempts to send the error messages and multiplying the total by 100. By computing this parameter, one may assess the ability of the Orchestration Manager to send error messages to the staff.

Reasoning About Food Placement:

▪ Analysis of the table layouts:

NB: for these tests, we divide the table in squares of 1 cm of side.

- Percentage of occupancy grid correctly analyzed (with respect to the fact that the squares are free or not): computed by increasing the counter of squares correctly guessed if the reasoning component guess correctly the state of one square (A square is correctly guessed if, when the reasoning component tells that the square is empty, it really is on the real table, and, if it tells that it is occupied, it is at least partially occupied on the real plate). We then divide the number of squares correctly guessed by the number of square of the table and multiply the total by 100. The computation of this parameter allows one to quantify the accuracy of the table analyzing part of the Reasoning About Food Placement component. The closer this parameter is from 100%, the more accurate the analyzing part of the Reasoning About Food Placement component is.

- Percentage of occupancy grid correctly analyzed (with respect to the fact that the squares are part of a free spots or not): computed by increasing the counter of squares correctly guessed if the reasoning component guess correctly the state of one square (A square is correctly guessed if, when the reasoning component tells that the square is part of a free spot, it really is on the real table, and, if it tells that it is not part if a real free spot, it is at least partially occupied on the real plate). We then divide the number of squares correctly guessed by the number of square of the table and multiply the total by 100. The computation of this parameter allows one to quantify the accuracy of the table analyzing part of the Reasoning About Food Placement component in determining free spots. This parameter allows one to verify that small free areas of the table are not seen as free spots by the Reasoning About Food Placement component. The closer this parameter is from 100%, the more accurate the analyzing part of the Reasoning About Food Placement component is.

- Percentage of optimal placement correctly determined: computed assuming that the free spots to place the plate are correctly determined by the Reasoning About Food Placement component (this parameter has no meaning if that is not the case). First we determine the common area between the optimal placement area determined by the Reasoning About Food Placement component and the one computed in the real world. Then we divide it by the area of the plate and multiply it by 100. The closer to 100% this parameters is, the more efficient is the Reasoning About Food Placement component ability to locate the optimal placement for the plate.

▪ Analysis of the determination of the risks:

- Percentage of situations where the risks have been guessed correctly: provided by the optimal placement of the plate and the different object on the table for different situation, we compute the number of times the risk has been correctly determined by the Reasoning About Food Placement component. We then divide this number by the numbers of attempt to determine the risk and we multiply it by 100. By computing this parameter, one may assess how skilled is the Reasoning About Food Placement component to accurately determine the risks in a real situation (the closer this parameter is from 100% the better the ability of the Reasoning About Food Placement component to determine accurately the risks).

▪ Analysis of the message transmission part of the component:

-Percentage of situations where the messages containing the optimal placement for the plate have been correctly sent: computed by dividing the number of times the messages have been correctly sent by the numbers of attempts to send the messages and multiplying the total by 100. By computing this parameter, one may assess the ability of the Reasoning About Food Placement component to send the messages containing the optimal position for the plate.

Order Verification and Error Handling:

▪ Spatial analysis:

-Percentage of times the component determines correctly if the plate is correctly grasped: computed by taking the ratio between the number of times the Order Verification and Error Handling determines correctly if the plate is correctly grasped and the number of times this functionality is being tested, the total multiplied by 100. By computing this parameter, one can assess the ability of the Order Verification and Error Handling component to perform with reliability the verification of the result of a grasping operation.

-Percentage of times the component determines correctly if the plate is correctly laid on the table: computed by taking the ratio between the number of times the Order Verification and Error Handling determines correctly if the plate is correctly laid on the table and the number of times this functionality is being tested, the total multiplied by 100. By computing this parameter, one can assess the ability of the Order Verification and Error Handling component to perform with reliability the verification of the result of a placement operation.

▪ Sound analysis:

-Percentage of times the component determines correctly if the client has a problem or not when discussing with him: computed by taking the ratio between the number of times the Order Verification and Error Handling determines correctly if the client has a problem or not and the number of times this functionality is being tested, the total multiplied by 100. By computing this parameter, one can assess the ability of the Order Verification and Error Handling component to perform to interact with the clients.

4)The description of the integration testing KPIs and the obtained result