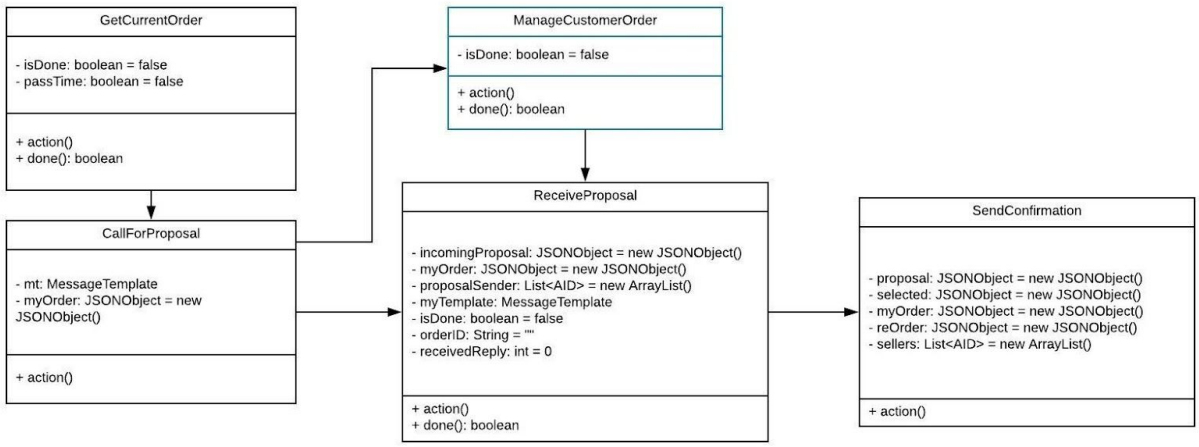
Customer Agent

Overview

The project simulates the interaction between customers and bakeries. The scenario file is given in the config folder which includes customer names, order id, time of order, and product list. Each customer gets the list of orders that belong to them and send the order at a specific time. The project is a part of the bigger project (class project) and it only manages the communication exchange between customers and order processing dummy. The order processing is supposed to be the part of bakeries that is responsible by taking orders and distributing it to other part of bakeries. However, here, the order processing dummy will only take order and reply it with the price of the each product.

Architecture



How It Works

A dummy of order processing is made to test the customer agent. The customer will have total five behaviors. The first four behaviors are managing message exchange between customer agent and bakery. The bakery here is the order processing agent. The last behavior manages the information given by mailbox to confirm the order has been arrived.

How to Run

Start.java adds all agents to be called including TimeKeeper, Customers and OrderProcessings. To run it, type “gradle run” in the terminal.

Behaviors

- **GetCurrentOrder**
  - Take time information (current time) from TimeKeeper.
  - Find order that has ‘time of order’ same as the current time.
    → If found matched, call the next behavior CallForProposal.
    → If no match found, inform bakeries that no order to be sent at the moment. This step is only executed for when the customer agent is interacting with class project.

- If the current time is more than the latest order based on the config file, shutdown the agent.
- If the number of order at the current time is more than one, behavior CallForProposal is called as many as the number of order.
- The behavior is restarted without waiting the whole message exchange with bakeries is finished.

- **CallForProposal**
  - The behavior is responsible on sending the current order (Out-2) to bakeries.
  - After sending the message, it calls the next behavior ReceiveProposal

- **ReceiveProposal**
  - The behavior waits for messages from bakeries.
  - The message received can be either a proposal or rejection.
  - The proposal message will include the name of the respective bakery and the price they offer for the order.
  - The behavior doesn't manage the content of the rejection message.
  - All proposal from bakeries are concatenated.
  - It waits until all bakeries give their answer.
  - If there is at least one proposal, it will call the next behavior SendConfirmation.

- **SendConfirmation**
  - The behavior starts with managing proposals received and find the cheapest offer.
  - It will then send the selected bakery, the updated order with list product that is the cheapest compared to other bakery.
  - It will also send the not-selected bakery a rejection.

- **ReceiveOrderConfirmation**
  - The behavior receives message from mailbox agent which is the information that all order has arrived.

Messages

- **Out Messages**
  1. Message to inform order processing agent that the customer doesn't have any order currently
     - **Performative**: INFORM
     - **Sender**: customer agent
     - **Receiver**: all order processing agent
     - **Content**: string
  2. Message to send order at a specific time
     - **Performative**: CFP
     - **Sender**: customer agent
     - **Receiver**: all order processing agent
     - **Content**: {"order_date":{"hour":4,"day":1},"delivery_date":{"hour":21,"day":2},"guid":"order-001","location":null,"customer_id":"customer-001","products":{"Donut":0,"Bagel":4,"Berliner":2,"Muffin":8,"Bread":2}}

        - **Conversation-id**: order.guid
    3. Message to send specific order type to the cheapest bakery
        - **Performative**: ACCEPT_PROPOSAL
        - **Sender**: customer agent
        - **Receiver**: (cheapest) order processing agent
        - **Content**: {"order_date":{"hour":4,"day":1},"delivery_date":{"hour":21,"day":2},"guid":"order-001","location":null,"customer_id":"customer-001","products":{"Donut":0,"Muffin":8,"Bread":2}}
    4. Message to reject proposal from expensive bakery
        - **Performative**: REJECT_PROPOSAL
        - **Sender**: customer agent
        - **Receiver**: (expensive) order processing agent
        - **Content**: string

- In Messages
    1. Price proposal from bakeries that are available
        - **Performative**: PROPOSE
        - **Sender**: order processing agent
        - **Receiver**: customer agent
        - **Content**: {name: "Sunspear Bakery", products: {"Donut":"4.54","Bagel":"3.21","Berliner":"2.94","Muffin":"4.43","Bread":"4.7"}}
    2. Refusal from bakeries if they can't process the order
        - **Performative**: REFUSE
        - **Sender**: order processing agent
        - **Receiver**: customer agent
        - **Content**: -
    3. Information from mailbox that the order has arrived
        - **Performative**: INFORM
        - **Sender**: mailbox agent
        - **Receiver**: customer agent
        - **Content**: -

Functions

- Retrieve(config_file)
  The function takes the content of the chosen config file and put it into a JSONObject for further processing.

- getOrder(customer_id)
  The function takes the order only for the respective customer agent.

- whenLatestOrder()
  The function sorts all time of orders and return the latest one as the latest order.

- findTheCheapest(proposal, order)
  The function compare each proposal and find the cheapest from all to update the product list in order.

- getCurrentOrder(current hour, current day)
  The function matches the current time with the order list and return the order that has the same time as the current time.

- includeLocation(order)
  The function includes location into the order. The location is available in config file however it is outside the order. Therefore we need to include it as it is important for the bakery to deliver the order.

Objects
Beside the behaviors and the agent itself, customer agent use class Date. This is a simple class to manage time of order. Comparator of Date then later be sorted to get the latest order.

```
public static class Date {
        public int hour;
        public int day;

        public Date(int hour, int day) {
                this.hour = hour;
                this.day = day;
        }

         public int getHour() {
                return this.hour;
         }

         public int getDay() {
                return this.day;
         }
}

Comparator<Date> comparator =
Comparator.comparingInt(Date::getDay).thenComparingInt(Date::getHour);
Stream<Date> DateStream = date.stream().sorted(comparator);
```