Restaurant Simulator

Marco Raggini - marco.raggini2@studio.unibo.it

August 2025

Contents

1 Introduction

RestaurantSim has the goal of improving the knowledge about multi-agent programming and multi-agent systems, in particular, the usage of Jason, an interpreter for an extended version of AgentSpeak.
This project aims to simulate a restaurant, implementing the behaviours of customers, waiters, and chefs.

1.1 Agent goals

In this simulation, three types of agents are represented:

- **Customer**: they ask to the waiter if there are any free tables; if not, the customer waits in a queue for his turn. Once the customer has taken a sit, they think for a few seconds, order to the waiter and then wait for the dishes to arrive. They take their time eating (e.g. a few second) and eventually they leave.
- **Waiter**: the main task carried out by the waiters is sitting the customers that ask for a table; if all the tables are taken, the request will be refused. Other duties are going to the tables, taking the orders and pass them to the chefs, bringing the dishes to the customers and then bringing them back to the kitchen once the customers are done eating.
- **Chef**: they take orders that arrive from the waiters and their task is to cook them in a settled period of time (the amount of time is based on the menu of the environment); eventually, when the dish is ready, they call the waiter to bring the dish to the table.

1.2 Agent properties

As we saw during the lessons, agents are defined as autonomous systems. There are many definitions of 'autonomy' in different contexts, and also the multi-agent systems context has one.
In order to call an agent autonomous, some properties have to be satisfied. The following list explains these properties and where they could be found in this project:

- **Pro-activity**: Agents generate their objectives, and attempt to achieve them; agents encapsulate control, and the rule to govern it. They are not driven solely by stimuli, they take the initiative and make something happen, rather than waiting for something to happen. In this project, customers are active entities, taking initiative by calling a waiter or ordering a dish.
- **Reactivity**: A reactive agent decides his activity from the interaction with the environment and from its changes. A purely reactive agent decides what to do without taking into consideration his history and he bases

his decision only on the present. All three agents can be considered as reactive: customers wait in queue until a table is free, waiters serve dishes only when they are ready and chefs can start cooking only when an order arrives to them.

- **Situadeness**: In this case, an agent is strictly coupled with the environment where it lives and interacts with it by his actions. Every agents in this project is stricly coupled to the environment, a single type of agent couldn't perform his goals without interacting with the environment.
- **Social Activity**: Social activity is the ability of an agent to interact with other agents. Cooperation, collaboration and also competition is considered a social activity. Agents in Restaurant Simulator have a lot of social activity; customers collaborate with waiters in order to satisfy them, waiters collaborate with chefs in order to prepare and bring the dishes to the customers, waiters compete with each other to serve more customers than the others, and so on.

2 Requirements analysis

This analysis takes into account some functional aspects that were not discussed in the introduction and all the properties that this software must have (non-functional).

2.1 Functional requirements

- **Customers**:
  1. When a customer arrives to the restaurant, he can be served by any waiter.
  2. Once the customer is brought to the table by a waiter, he will refer only to that waiter.
  3. Once the customer is sat at the table, he must choose randomly a dish from the menu.
  4. If all tables are occupied, the customer has to wait in the queue.
- **Waiters**:
  1. A waiter can only perform one task at a time. If he is busy, he must communicate it.
  2. If a table is free, the waiter must check if someone is waiting in the queue and, in that case, bring him to the table.
  3. Waiters have to respect the order of the queue (they can't chose a random customer in the queue).
  4. When a waiter takes an order, he has to give it to a randomly chosen chef in the kitchen.

- **Chefs**:
  1. When an order arrives to the chef, he must start cooking.
  2. Chefs have to follow the order of the orders given by the waiters.
- **Environment**:
  1. The environment must manage the tables and their state.
  2. It must store the menu and give it to the customers.
  3. It must perceive the queue to the waiters.
  4. The environment has to manage the competition between waiters. An example could be: two waiters try to occupy the same table at the same time with two different customers; in this case, only one waiter 'wins'.
- The system must has a graphical interface to visualize changes in the environment, the queue, the tables and the movement of the agents. It also includes a logger of the actions that can't be seen in the graphical interface for a better comprehension of what is going on.

2.2 Non-functional requirements

- Scalability:The system must work also for high numbers of agents.
- Comprehensibility: The system and the code must be easy to understand for external people that need to analyze them.
- Testability: The system must be tested. The tests must be clear, easy to implement and easy to run.
- Parallelism: The system must manage the concurrency between agents and the fact that the waiters serve many tables at the same time as well as the chefs who cook different dishes at the same time.

3 System architecture

The system architecture is composed of two main folders that divide the agents from the environment.

- asl: it contains all the codes written in **Jason**, the agent-oriented programming language used to define the behaviors of the intelligent agents within the system. More specifically, it includes the files `customer.asl`, `waiter.asl`, and `chef.asl`, each one representing the reasoning and behaviour of a different type of agent in the simulation. These files define the agents' beliefs, goals, and plans, and form the core of the multi-agent interaction logic.

- java: This folder includes all the Java classes that define the environment in which the agents operate. It contains the implementation of the environment's internal logic, the graphical user interface (GUI) and the test files used to validate the software.

This separation allows a clear distinction between the agent logic and the environment dynamics, making the system easier to maintain and extend.

The environment architecture is designed by using the pattern MVC (model-view-controller). This pattern consists of the separation of concepts, dividing them into three main areas.

The core area is the Model, which represents the data and the logic of the application. It's responsible for managing the data, responding to requests for information, and updating itself based on instructions.

The View concerns the user interface. It is the graphical representation of the model; it takes all the data from the model, displays them, and updates the interface when it is necessary.

The Controller is an intermediary between Model and View. Its main job is to enable communication between the two parts; it enables the real separation of what is about logic and what is about view.

This architecture allows a clear division between different parts of the program. The Model and the View have their own folder, called `model` and `view`; instead, the Controller doesn't have its own folder because it is represented by a single file called `RestaurantEnvironment`. This file permits the communication between Model and View, but it is also the representation of the environment for the agents. When an agent has to change something in the environment or gets percepts of it, this file manages all these behaviors.

3.1 Entities

This subsection should explain which are the entities, their behaviour, and how they are implemented. An important note to mention before the explanation of the entities regards the movement. In this project, the movement is not implemented but is simulated. This means that the agents are not moving cell by cell but they 'teleport' from the starting position to their destination. As explained in the introduction, the movement is not part of the project, but it is important for a better comprehension, in particular for the graphical interface. The movement is implemented by calling a method of the environment that calculates the position where the agent will be teleported.

- **Customer**: Customer is initialized as an agent with three beliefs: `customer_state(free)` that represents the current state of the customer, which it can be `free` or `sitting`. Then there is `assigned_table(none)`

that is used to store the information of the table in which the customer is seated and `waiter_to_call(none)` that stores the information of the waiter to call.
When a customer spawns, they wait a random fraction of time between 0 and 1 second (to simulate a real scenario where all customers don't arrive at the same time). After waiting, they try to find an available waiter. In order to do that, the customer asks every waiter for his state and the first waiter to reply with a `+waiter_available` is the one who will be called to have a table.
It's possible that a customer asks for an available waiter to get a table, but all tables come out already occupied. In this case the waiter tells the customer to go into the queue. The customer will go in the queue and wait until a waiter sends him a `+your_turn` that allows him to ask again for a table.

When the customer sits at the table, his beliefs are modified: its current state changes from `free` to `sitting` and a value is assigned to `assigned_table` and also to `waiter_to_call`. Then, when he randomly chooses a dish by perceiving the menu from the environment, he orders it from the waiter and he waits until `+dish_arrived(D)`. Once the dish has arrived, the customer eats and then he leaves the restaurant by calling a function of the environment that frees the table.

- **Waiter**: The waiter starts with three beliefs: `waiter_state(free)`, `chefs_available([])`, which is the list of chefs who can be called for assigning an order and the `customer_queue([])`, that represents the customer queue perceived from the environment. There is also an another belief which he faces during his life that is `table_status(T, State)`, which is created by the waiter to remember the states of the tables that he serves. The waiter has no active goal while he has been spawned, but he reacts from a large variety of stimulus:
  - `+new_queue(Queue)`: When the waiter perceives a new queue from the environment, he substitutes his queue with the new one.
  - `+chef_available[source(Chef)]`: The waiter adds the new chef to his list.
  - `+table_status(T, Status):table_status(T, free) & customer_queue(Q)`: When a table frees, the waiter checks if there are customers in the queue. In this case, the waiter calls the first one by telling `.send(Customer, tell, your_turn)` and he removes him from the queue.
  - `+?waiter_state[source(Customer)]`: It is used by the customer to get to know the state of the waiter.
  - `+!called_for_a_table[source(Customer)]`: This goal is called by a customer who wants to sit at a table. There are many scenarios for this goal: if the waiter is free and there is almost one table free,

the waiter changes his state to busy and he tries to assign a table to him. When the table is assigned to the customer the waiter brings him to the table and tells him the `TableId`. If the waiter is free but the tables are all occupied, he sends the customer into the queue. If the waiter is busy, he replies to the customer to call him later.

  - `+!take_order(Dish)[source(Customer)]`: This goal is called by a customer who wants to order a dish. In this case, the waiter becomes busy, takes the order, chooses one of the chefs in the `chefs_available` list randomly and then sends the order to him. Once he is done, he changes his state to free again.
  - `+!dish_ready(D, T)[source(Chef)]`: When a dish is ready and the waiter is free, he brings the dishes from the chef to the right table.

- **Chef**: The chef is initialized with one belief: `order_queue(none)`, which is implemented when the agent spawns by using the standard function of Jason called `.queue.create(Q)` that creates an empty queue called **Q**. The chef's behaviour is simple, when he spawns, he tells to every waiter his existence. Then he checks the queue: if it's empty he waits and then checks again, if he has any order, he takes the first order and remove it from the queue by calling `.queue.remove(Q, Order)`. Every order has three fields: the `Dish`, the `TableId` and the `Waiter` who sends the order. On the other hand the dish is composed of the name and the preparation time. Once the chef has taken an order, he checks the preparation time of the dish and starts cooking, when he finishes, he calls the waiter and informs him that the order for the specified table is ready.

- **Table**: The table is not an agent in this system, but fundamentally it's an entity. It is managed by the environment with a list of tables and their states. In particular, the environment manages the concurrency between agent who wants to change the table's state by implementing a lock. The table is represented in the model by a class called `Table.java` which contains the ID, the state and the position, which calculated at the start of the program and is based on how many tables there are and how big is the restaurant.

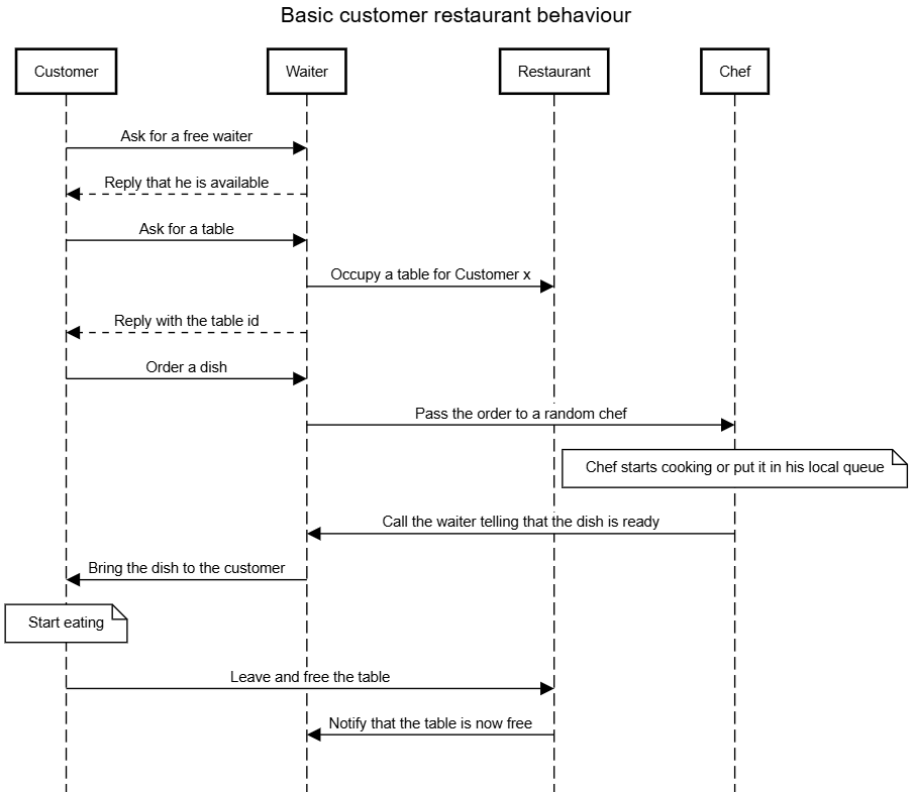
- **Restaurant**: It represents the environment where the agents act. The restaurant is implemented in Java with the class `RestaurantImpl.java`. This entity manages all the interactions between agents, the queue or agents and the tables. Managing these interactions leads to encountering concurrency issues, as the agents can act on the same entities at the same time. These issues were handled through locks and synchronized functions. At the logic level, the restaurant only stores the information about the tables, the queue, and their implementation. In reality, however, it also includes the logic for the movement of the agents. This functionality is implemented in the environment because movement is not a constraint

but rather a feature added for graphical purposes and to improve understanding. In particular, some movement and graphical functions are implemented as followed:

- **Placing tables**: At the start of the program, tables are placed in the environment in order to be equally distant one to each other, based on the width of the restaurant.
- **Placing chefs**: At the start of the program, chefs are placed in the south-left corner, equally distant.
- **Add customer into the queue**: The queue is ordered by FIFO (First In First Out). This rule is followed also in the graphical part, where the view is updated every time there is a change.
- **Agent's movement**: Every movement is performed by calling the relative function like: `go_to_queue`, `go_to_table`, `go_to_chef` and `go_to_default_position` (when a waiter is free and has nothing to do, he goes in the "rest" place, where he waits until the next order).

3.2 General behaviour

The environment presents various situations to which the agents must adapt, but it is important to keep the base case in mind in order to better understand the issues that may arise. The diagram below illustrates the general behaviour of the system.



3.3 Faced issues

During the development of the program, several issues were faced. The majority of them concern the concurrency between agents or between the agents and the environment. In particular, the problems between agents occur when, for example, a customer calls a waiter who is busy because he is perceiving another goal. In this case, the waiter notifies the sender of the message that he is busy and tells him to try later. When the customer receives this message, waits a random period of time and retries once the time is up.
In general, every action that creates concurrency problems between agents is managed using the method 'Wait and retry'.

On the other hand, the concurrency between agents and the environment is managed by the environment itself. In Java, there are more functions and libraries for managing these situations. In the model, every function that uses or modifies the data structures is surrounded by a `synchronize`. That function doesn't allow many threads to run the same codes at the same time.

4 How to run projects

Gradle and its own wrapper are used in order to manage the dependencies and run the program in a simple way. Thanks to the Gradle wrapper, it is possible to run the project simply by calling the script `./gradlew run` or `./gradlew runRestaurantMas`.
Once the program has started, two panels are opened: the Jason console that permits to visualise every log printed by the agents and also the buttons to stop, pause, and continue. The second panel is the graphical interface, where it is possible to see the restaurant and the agents at work. Near the graphical interface, there is another log area, with fewer logs, that allows to understand all the actions performed by the agents which are impossible to see in the graphical part: a dish a customer is ordering from the waiter, a dish a chef is preparing, etc. In the view four colors are displayed: **Brown** is a table, **Red** is a chef, **Yellow** is a waiter and **Green** is a customer. The queue is represented in the north-right corner of the screen.

4.1 How to run tests

The tests follow the same logic as the real program. To run the tests, use the script `./gradlew test`.

5 Future development

During the development of the system, several potential improvements were identified, which could be explored in future work.
The first one is the introduction of a new agent called **Master**. His goal will be the management of the other agents in terms of spawning and killing the agents. This could be important to make the program more dynamic. When a customer has finished eating and leaves the restaurant, the master can 'kill' the agent in order to stop him, and he can also spawn randomly new customers that want to sit at a table.
Other possible improvements include adding new behaviours, such as: the customer asking for the bill, customers arriving at the restaurant as a family and being able to sit at the same table, the chefs having a single queue for the dish to prepare meaning that, when a chef is free, he checks the next dish to prepare and removes it from the queue.