

FlyPizza: Autonomous Drone Delivery System

(v. 0.1.0)

Davide Di Marco
`davide.dimarco5@studio.unibo.it`

October 30, 2024

The project proposes the development of a MAS (Multi-Agent System) using the JASON programming language to simulate a drone delivery system for a pizzeria. The aim is to create a realistic environment composed of several agents: drones, a pizzeria, and a rescue robot. The simulation include a map with obstacles. The drones are autonomous agents capable of navigating, avoiding obstacles, and making decisions based on their beliefs. The overall delivery system is centrally controlled by the pizzeria. A user interface is also provided to visualize the environment and observe how the system works. This project aims to simulate an autonomous drone delivery system demonstrating the agents' ability to perform tasks independently.

1 Goals/requirements

The goals of this project is to provide a simulation of automated drones delivery systems for a pizzeria. We can imagine a pizzeria with some drones able to delivery pizzas to certain destinations.

The main goal of the project is deliver pizzas at destinations. To be reached, the main goal can be split into multiple requirements:

- **Dynamic drone navigation:** Drones must dynamically navigate the map, avoid obstacles, and adjust routes autonomously.
- **Collision avoidance:** The system must ensure drones can detect and avoid collisions with both static and dynamic obstacles.
- **Battery management:** Drones should monitor their battery levels, make decisions about when is necessary to charge.
- **Power engine selection:** Different power modes for the drones should be considered depending on the distance of the destination.

- **Pizza assignment policy:** The pizzeria must assign deliveries to available drones based on their state. Drones must be ready to deliver considering their properties as battery charge, broken state etc...
- **Rescue robot policy:** In case of drone malfunctions, a rescue robot should be deployed to retrieve the failed drone and return it to the pizzeria to repair it.
- **User interface:** An interface will be provided to monitor the drone system and observe how drones interact with the environment.

1.1 Question and Answers

Here are some questions to help clarify the project requirements, particularly focusing on the agent interaction with the system and describing some technical requirements that must be met.

- **Question:** *How orders are generated?*
Answer: Orders are generated at random time every certain number of seconds by the pizzeria itself.
- **Question:** *How delivery assignment is managed?*
Answer: The delivery assignment is managed by the pizzeria agent that assigns the delivery to drones basing on battery charge, orders queue and drone availability.
- **Question:** *How the drone's battery is managed?*
Answer: Every step drains a certain amount of battery. The battery drain depends also from the engine power selected by the drone to reach the destination.
- **Question:** *What happens in case a drone is broken or unavailable?*
Answer: If the drone is broken the robot starts the recovery, going to the drone position, take it and transport it to the pizzeria to be repaired.
- **Question:** *How many pizza can be trasported by a drone?*
Answer: Only a pizza per delivery can be trasported by the drone.

1.2 Scenarios

This simulation can help pizzeria managers see how drone deliveries might work in real life. It can also be useful for other businesses that need to make deliveries, like package shipping or delivering medical supplies. The project includes an easy-to-use interface where users can watch at the delivery process in real-time. They can see how the drones move, avoid obstacles, and complete their deliveries.

1.3 Self-assessment policy

To ensure the FlyPizza system meets functional requirements, the following self-assessment strategies are implemented:

JUnit for Java model testing JUnit [4] is used for automated unit testing of the Java model components. The model is used to update the environment percept by the agents, so it is important to test the proper logic of the model.

Logger and debugger (JASON) Jason include a built-in debugger and logger. It is used to track the behavior of the system. It is useful to track all the events that happens in a certain time.

Functionality testing Each functionality must meet the requirement so the system is tested using different configuration with one, two and more drones with manual test the functionality troubleshooting logs to escape bugs.

2 Background

In order to understand the motivations and development choices behind the "FlyPizza" project, it's important to introduce some key theoretical concepts and technologies that are being used during the project.

2.1 MAS - Multi-Agent System

Flypizza is based on Multi-Agent System (MAS). A MAS is composed by one, but typically more, agents. An agent, in this context, is an entity capable to make decisions based on the perceptions and the beliefs, that produces an internal information, capable to change the behavior of the agent. Perceptions and Beliefs can be received from the environment where the agent is introduced or by other agents. The agents works together to reach a common goal or individual goal and the failure of one of these can change the behavior of the agents to make remediation in order to reach the final goal also in case of failure. Agents also have computational capabilities, so they can manipulate data and not only execute actions based on beliefs and perceptions.

2.2 BDI Architecture

The BDI (Belief-Desire-Intention) architecture [1] is a popular framework for designing intelligent agents within a MAS. It provides a structured way for agents to make decisions and take actions based on their internal states and external stimuli. In FlyPizza, each agent (drones, pizzeria, rescue robot) is modeled using the BDI architecture.

- **Beliefs** represent the information an agent has about the world, including both its internal state and external environment. For example, a drone's beliefs may include its current battery level, location, broken state etc. These beliefs are dynamically updated based on perception and communications with other agents.
- **Desires** are the goals that an agent aims to achieve. In FlyPizza, a drone's desires could include successfully delivering a pizza to a destination, returning to the pizzeria for recharging, or avoiding collisions.
- **Intentions or Plans** are the plans that an agent commits to in order to achieve its desires. Once an agent adopts an intention, it focuses on executing the necessary actions to achieve that intention. For instance, a drone may intend to follow a specific delivery route, which involves navigating to the destination, avoiding obstacles, and returning to the pizzeria.
- **Decision-Making Process** The BDI model permits a decision-making process where agents continuously evaluate their beliefs, formulate desires, and commit to intentions. This flow allows agents to adapt to changing conditions. For example, if a drone detects a sudden obstacle on its path, it can update its beliefs, reassess its desires, and adjust its intentions accordingly.

Implementing the BDI architecture in FlyPizza offers several advantages:

- **Modularity:** Each agent's beliefs, desires, and intentions are encapsulated, making the system easier to manage and extend.
- **Flexibility:** Agents can adapt their behavior in response to environmental changes or internal state change.
- **Scalability:** The system can scale by adding more agents without increasing complexity.
- **Robustness:** Decentralized decision-making enhances the system's ability to handle failures or unexpected scenarios.

2.3 Used Frameworks / Technologies

This section explains the concepts and technologies involved in the development of the project, which explores MAS programming, BDI Architecture and OOP (Object-Oriented Programming) model.

- **JAVA:** is the language used to develop the model of the project and the interaction make possible the interaction with JASON [3]. Java also permits to create entity and organize code in object respecting OOP Paradigm.
- **JASON:** is an OpenSource interpreter for an extended version of AgentSpeak: the language used by JASON. Jason is particular indicated in projects where agents

need to make autonomous decisions depending on the environment [2]. Jason also support a user-interface to easily visualize the behaviour of the system and understand the choices of the agents: avoid obstacle, charge etc.

In FlyPizza drones, pizzeria and rescue robot are agents that cooperate and collaborate to ensure that the pizzeria ends the orders to deliver in an efficient way. This set of information composed by beliefs, goals, and plans are managed using the extended version of agentSpeak: JASON.

3 Requirements Analysis

By examining the requirements described earlier, we can identify several essential functions that the system needs to have to meet the specifications listed in the previous section. In particular we identify some non trivial aspects that are described below.

Functional Requirements

- **Order Assignment Policy:** The system must ensure that pizza delivery orders are assigned to drones in a logical way and not in disorder.

Solution: Implement an orders queue. Orders are placed in this queue when generated, and the pizzeria agent is responsible for distributing them from the queue to available drones. The pizzeria agent evaluates each drone's current status and battery level. The use of a queue ensures that orders are processed in the order they are received, while also allowing the pizzeria agent to reassign orders if a drone is unavailable. This centralized approach introduces a communication channel between the pizzeria agent and drones.

- **Drone Delivery:** The entire process of delivery of pizzas must be coherent with the system's state considering battery charge, obstacles, and the availability of drones.

Solution: Use a centralized decision-making approach where the pizzeria agent assigns orders based on available information about the environment and drone statuses. The use of a centralized introduce the message exchange between pizzeria agent and others (drones and robot).

- **Robot Recovery:** In case of a drone failure, the system must have a mechanism to recover the drone and bring it to the pizzeria.

Solution: Use a robot agent in the FlyPizza system. The centralized pizzeria monitors drone failures in real-time using messages exchange between drone, pizzeria and robot. Soon as a failure is detected, the rescue robot is deployed.

- **User Experience:** The user experience must be coherent with the state of the environment and real-time synchronized. Furthermore the user must have a way to verify the coherent status of the system.

- **Solution:** Any updates to the environment like drone locations, are immediately shown in the user interface and the logger shows the actions at a certain time.

Non-Functional Requirements

- **Timing Synchronization:** The order of the operation must be synchronized. **Solution:** Include conditions for certain plans that could be not synchronized. Include conditions avoid the facts that certain information and perceptions can be received not in a proper time.
- **Remediation to Failure:** In case of failure the system must react to failure executing some remediation. **Solution:** Also in this case include some control flow to ensure that all works in a proper way. For example if the drone's battery fall down unexpectedly due to not consistent state, the system must recognized it and apply some remediation: for example the recover of the drone.
- **Information flows, message protocol between agents:** The information flows between agents in this type of projects is very important. It's not the same thing say "Pizzeria say X to drone" and say "Drone send a message Y to Pizzeria" **Solution:** This requirements can be analized in different ways. The approach we prefer it's to think about the real environment and how protocol communications works in a logical way. The communication flows must be simple as possible and reflecting as much as possible the responsibilities in a real world.

4 Design

After analyzing the requirements, we proceed to illustrate the design approach adopted for the development of FlyPizza. Given that the system is designed as a Multi-Agent System (MAS), we have chosen to implement the BDI (Belief-Desire-Intention) Architecture.

In our workplan we decide to divide the design phase in two parts:

The **first part** of the design focuses on the agents. Each agent in the system is designed with specific responsibilities. During this phase, we define their initial beliefs, their desires, which represent their objectives or goals, and their plans, which are the actions they will take to achieve those goals. At a higher level, we also determine the overall responsibilities of each agent within the system and some non-trivial aspects such the communication between them and a proper way to make the communication flow from an agent to others.

The **second part** of the design clarify the model used by the system. This model is organized into Java classes. The use of object-oriented programming allows us to divide the logic and the abstraction of the concept making the model easier to manage and extend.

4.1 Structure

Package Diagram The project is organized into packages, each containing components such as Java classes or ASL agent files that encapsulate the agent logic. As shown in Figure 1, the project structure follows principles of maintainability, separation of concerns and modularity. Each package is designed with a specific responsibility.

The structure is divided as follows:

- **asl**: This package contains ASL files that define the agent logic, separating them from the rest of the model code. This makes it easy to identify the agents, which represent the core of the system.
- **Java**: This package contains all the Java classes that serve as the backbone of the system. The internal structure of this package is clearly divided based on specific concerns:
 - **env**: This is the main package, including the model and the view. It is important because it identifies the main sections of the system.
 - **utils**: This package contains utility classes. In our case, it is used to create the class that defines InternalActions in Jason.
 - **model**: This package encapsulates logic model, divided into behavior and objects. The **objects** package contains the class definitions for entities, the **behavior** package contains the classes that describe the behavior as drones' movement and failure handling by the environment.
 - **view**: Responsible for the user interface. It manages the representation of the model within the jason provided view.

This design promotes a clean separation of concerns, allowing the system to be extended and maintained.

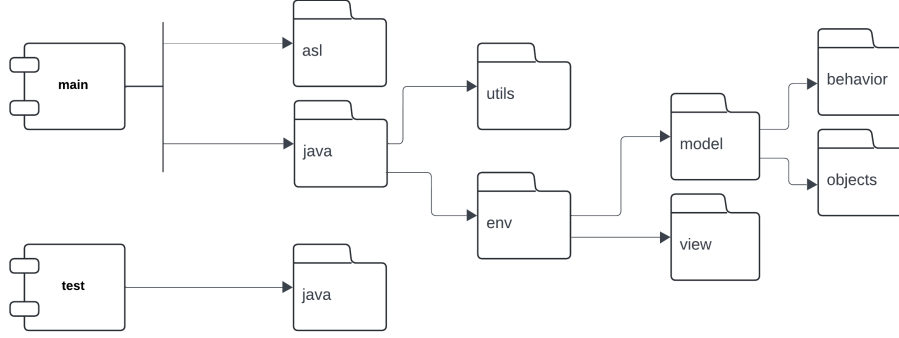


Figure 1: Package Diagram

Class Diagram The figure 2 shows the structure of the Model components through a UML class diagram, in which only the most significant fields and methods are shown for each entity.

The **Environment** entity encapsulates the core structure of the system, representing the overall environment in which the drones, pizzeria, robot, and obstacles interact.

The **FlyPizzaEnv** class inherits from the jason **Environment** and is responsible for handling the execution of the system. It contains key components such as a **Logger**, the **FlyPizzaModel**, the **FlyPizzaView** and an **ExecutorService** that manages task execution. This class includes methods such as `init(String nickname)`, which initializes the environment, and `executeAction(String ag, Structure action)`, which processes actions taken by agents in the system.

The **FlyPizzaModel** is crucial as it encapsulates the core logic of the system. It contains the grid size (`gSize`), a set of obstacles (`Set<Location>`), a list of drones (`List<Drone>`), a reference to the **Pizzeria**, a reference to the **Robot**, and a **NavigationManager** that handles the movement and navigation logic for the drones. The model also provides methods such as `addObjects(int nDrone)` for adding new objects to the environment and `moveTowards(Location dest, int agentId)` to control the movement of drones based on their destination and agent ID.

The **FlyPizzaView** class manages the graphical representation of the environment. It interacts with the **FlyPizzaModel** to draw the agents (drones, robot) and obstacles in the environment. The view includes methods such as `draw(Graphics g, int x, int y, int object)` for drawing elements of the environment and `drawAgent(Graphics g, int x, int y, Color c, int id)` for representing the agents visually on the user interface.

Considering the **Drone** entity, it represents the agents responsible for delivering pizzas. Each drone is characterized by an ID, its location on the map (**Location**), battery level, a unique drone name, and an **EngineMode**, which determines its current power mode (**FULL**

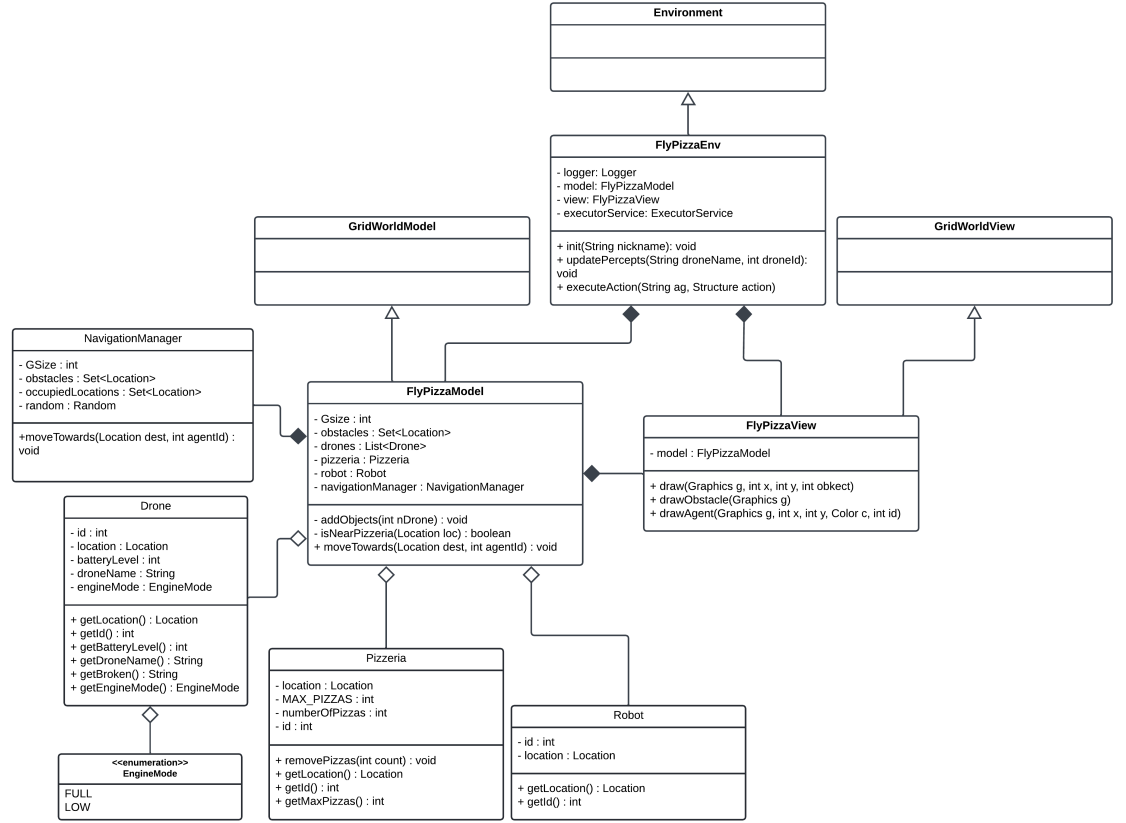


Figure 2: Class Diagram

or LOW). Drones possess methods to manage their movement (`moveTowards(Location dest, int agentId)`), retrieve information about their status (`getBatteryLevel()`), and handle engine mode `getEngineMode()`

The **Pizzeria** entity represents the central agent managing the pizza deliveries. The pizzeria is defined by a `Location`, a maximum pizza capacity (`MAXPIZZAS`), and methods like `removePizzas(int count)`, which decreases the pizza count when deliveries are made, and `getMaxPizzas()`, which retrieves the maximum number of pizzas.

The **Robot** entity handles the recovery of broken drones. It is defined by its ID and location and can perform operations like movement, rescuing and repair. The robot is essential for maintaining the system's functionality by retrieving malfunctioning drones and bringing them back to the pizzeria for repairs.

The **NavigationManager** is responsible for guiding the drones through the grid, avoiding obstacles and ensuring they reach their destinations. It tracks key attributes such as grid size, the positions of obstacles, and the occupied locations of drones. Its key

method, `moveTowards(Location dest, int agentId)`, calculates the movement path for a drone to reach its destination.

Lastly, the **EngineMode** enumeration defines the two power modes available for drones: **FULL**, which is used for long-distance deliveries requiring more power and speed up the delivery, and **LOW**, which conserves battery for shorter trips.

4.2 Behaviour

In this section, the behavior of the agents involved in the pizza delivery system is described using Activity Diagrams. Each agent (Drone, Robot, and Pizzeria) has specific plans that define its operation. The following paragraph provide an overview of the Activity Diagrams for each agent where all the implemetative details are excluded. The diagrams are described from the point of view of the agent itself.

Drone Activity Diagram Figure 3 describes the behavior of the drone, whose main objective is to deliver the pizza to its destination. During all the delivery the drone checks it's current perceptions to be coherent with the enviroment.

The process begins with the (**Receive Order**), triggered when the drone receives a new order from the pizzeria. After receiving the order, the drone prepares to move towards the destination, but before it starts moving, it is essential for the drone to check its status in order to make the appropriate decisions. So the drone checks if it is broken. If the drone is found to be broken, it sends a message (**brokenMessage**) to both the pizzeria and the rescue robot. If the drone is available and not broken, it retrieves its current position and checks whether the position data from perception given by the environment is valid. Next, the drone decides which engine mode to use, choosing between the two available options: **FULL** and **LOW**. To make this decision, the drone calculates the required battery level for **FULL** mode (the worst-case scenario) and compares it to the current battery level. At this point, the drone performs an additional battery check to ensure that it has enough power before starting the movement. Once these checks are completed, the drone starts moving toward the destination. At each step, it retrieves current position percepts (**Get Current Position Percepts**) to stay aligned with its own state and the state of the environment. It is important to note that the drone might break down due to low battery or random failures caused by environmental factors.

When the drone obtains its current position, it checks if it has reached the destination by comparing its current position with the target position assigned initially by the pizzeria. If the drone reaches the destination, it completes the pizza delivery and then returns to the pizzeria.

Upon returning to the pizzeria, the drone checks its battery status. If the battery level is below 50%, it recharges and sends a message to the pizzeria indicating that the charging is complete. Otherwise, if the battery is sufficient, the drone send a message to pizzeria agent to signals its availability for a new order assignment described by the **Send Message process Order queue** to pizzeria.

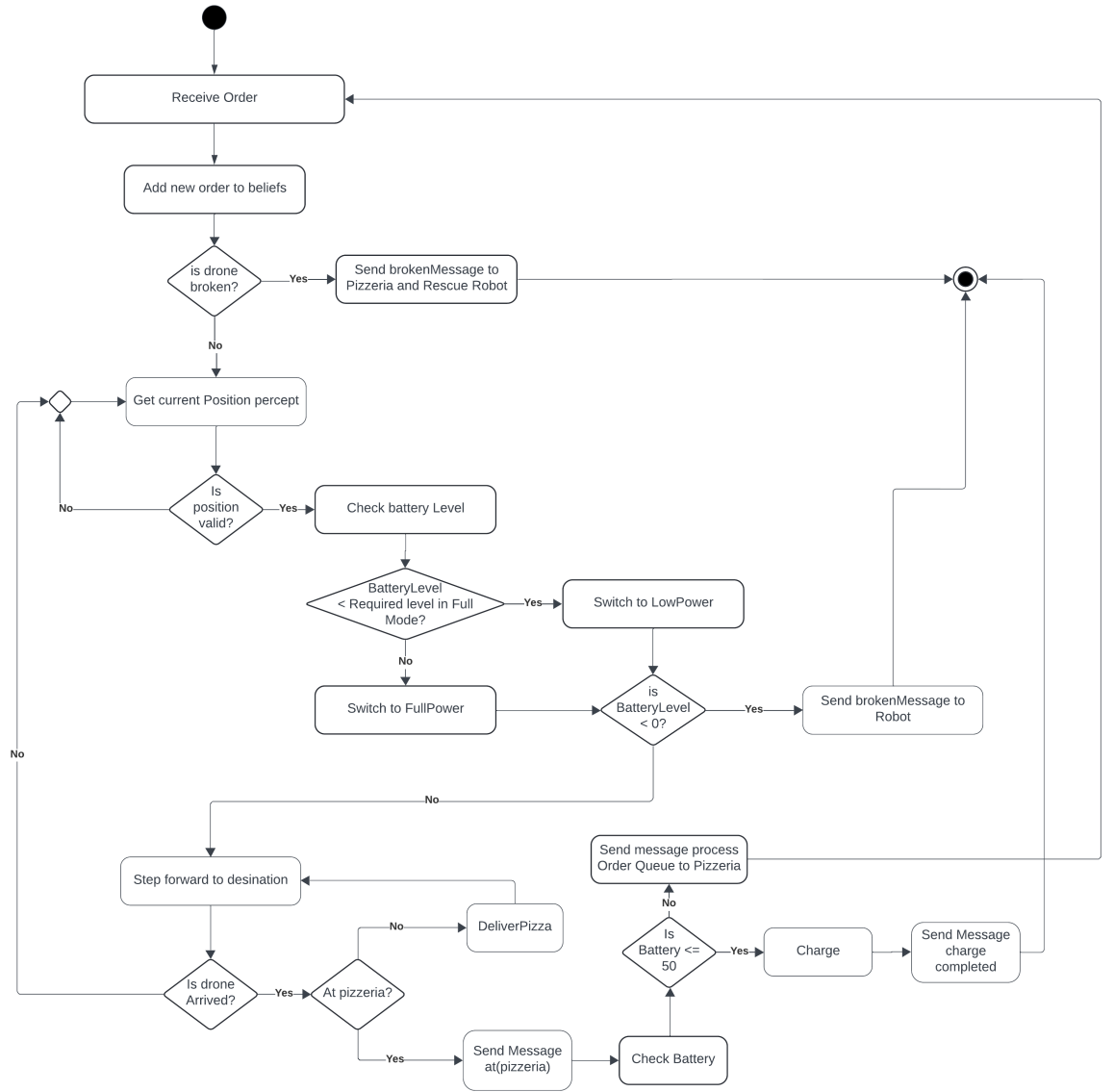


Figure 3: Drone Activity Diagram

Robot Activity Diagram Figure 4 describes the behavior of the robot, whose main objective is to recover broken drones and return them to the pizzeria for repair. The robot operates by processing each notification of a broken drone sequentially and ensuring its successful recovery. The process begins with the robot waiting for a broken

drone notification (**Wait for broken drone notification**). When the robot receives a notification, it checks if it is currently busy (**Is Robot busy?**). If the robot is already handling another broken drone, it adds the new broken drone to its perception for future handling. Otherwise, the robot marks itself as busy (**Set Busy**) and proceeds with the recovery process. Next, the robot starts moving towards the location of the broken drone (**Move To Location**). At each step, the robot checks if it has arrived at the location by comparing its current position with the target position (**Arrived At Location?**). If it hasn't reached the location, the robot continues moving until it arrives. Once the robot reaches the destination, it verifies whether it is at the pizzeria or at the broken drone's location (**Is at Pizzeria?**). If the robot is at the broken drone's location, it picks up the drone and starts moving back to the pizzeria (**Pick Up Drone then Move to Pizzeria**). Otherwise, if the robot is at the pizzeria, it proceeds to drop off the drone (**Drop Off Drone**) and initiates the repair process (**Repair Drone**). After repairing the drone, the robot checks if there are other broken drones in its perception (**Are There other brokenDrone perception?**). If there are, the robot proceeds with the next recovery; otherwise, it sets itself to not busy (**Set not Busy**) and returns to waiting for new notifications. This flow allows the robot to handle multiple drone recovery tasks, processing them one at a time.

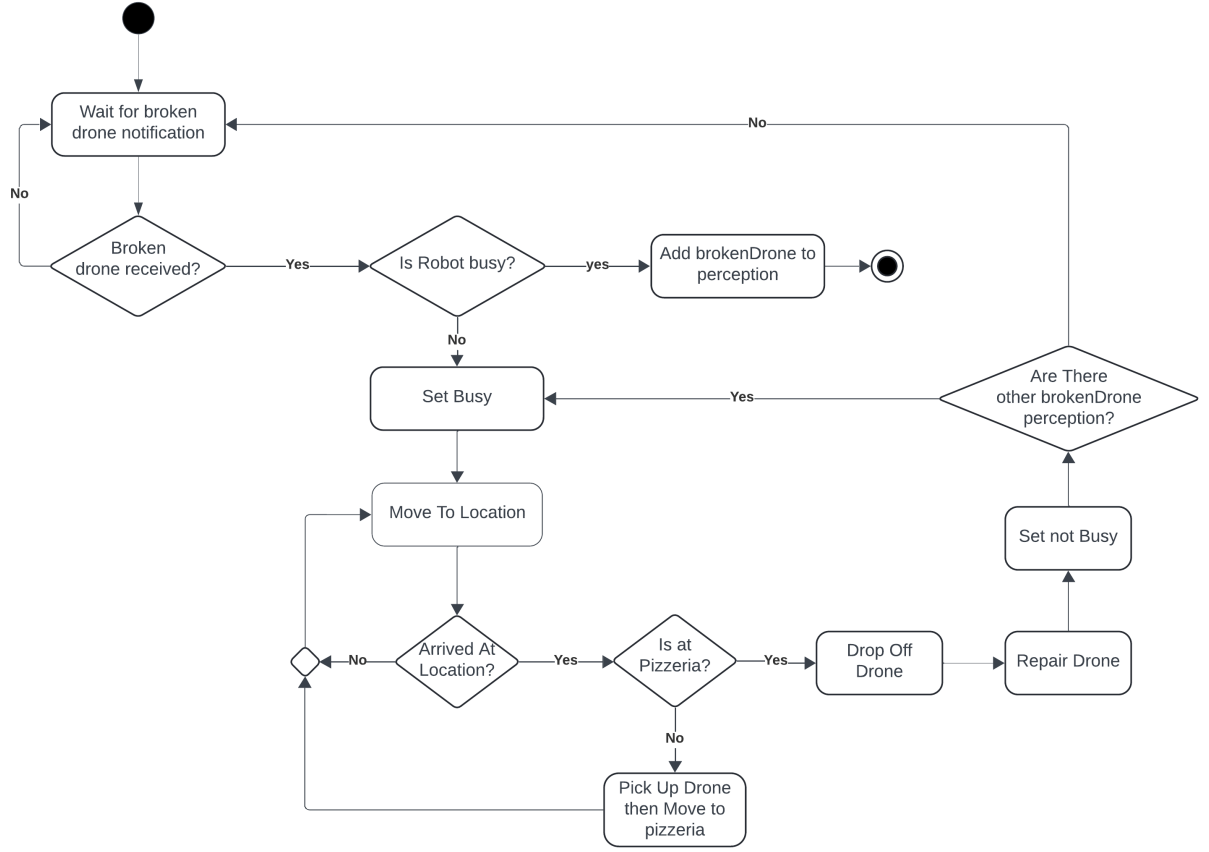


Figure 4: Robot Activity Diagram

Pizzeria Activity Diagram Figure 5 describes the behavior of the pizzeria, whose main task is to generate and manage pizza delivery orders and assign them to drones. The process begins when the pizzeria starts operating, waiting 5 seconds before generating orders (**Wait 5 seconds**). After this initial wait, the system starts generating orders (**Generate Orders**). Each new order is generated after a random interval of time (**Wait random time**) to simulate variability in order arrival. The system checks if the number of generated orders has reached the predefined maximum (**generatedOrders < NumberOfMaxPizzas**). If the limit has not been reached, the system generates a random destination for the order (**Generate random destination for order**), then adds the order to the end of the queue order (**Enqueue order at end**), and proceeds to process the queue (**Process Order Queue**). If the queue is not empty, the pizzeria starts dequeuing orders one by one (**Dequeue Order**) from the head and checks if there are any available drones to handle the delivery (**Check Available Drone**). To assign an order,

a drone must be at the pizzeria, not broken, not charging, and its battery level must be sufficient for the delivery. If a suitable drone is found, the order is assigned to the first available drone (**Assign order to first available Drone**). If no drone is available, the order is re-enqueued at the front of the queue (**Enqueue order at front**) to be handled later instantly when a drone becomes available. Once the maximum number of orders has been generated (**generatedOrders** \geq **NumberOfMaxPizzas**), the system processes any remaining orders in the queue (**Process Remaining Orders**). The system checks whether the order queue is empty (**Is Order Queue Empty?**) and, if there are still orders to be processed, it continues to assign them to available drones until the orders queue is empty.

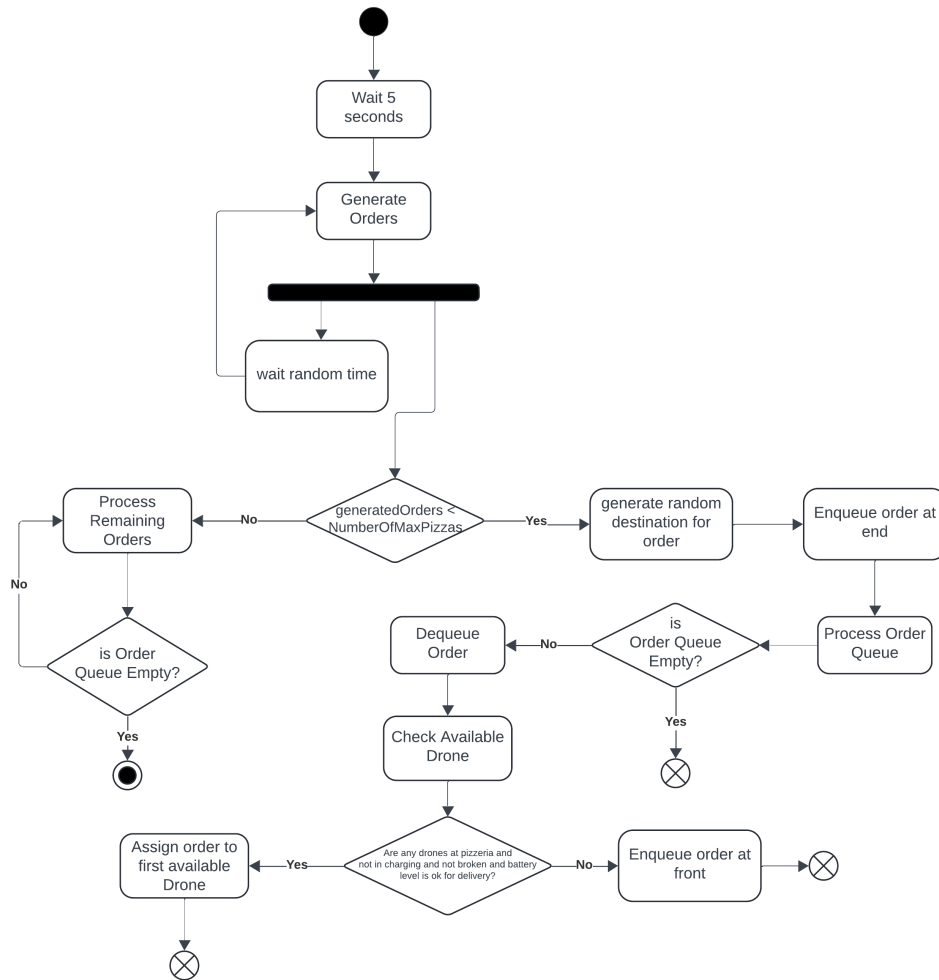


Figure 5: Pizzeria Activity Diagram

4.3 Interaction

In this section, we describe the interactions between the main entities involved in the pizza delivery system: the **Drone**, the **Pizzeria**, the **Robot**, and the **Environment**. These interactions are illustrated through sequence diagrams that present the flow of events and the messages exchanged between the entities of the system.

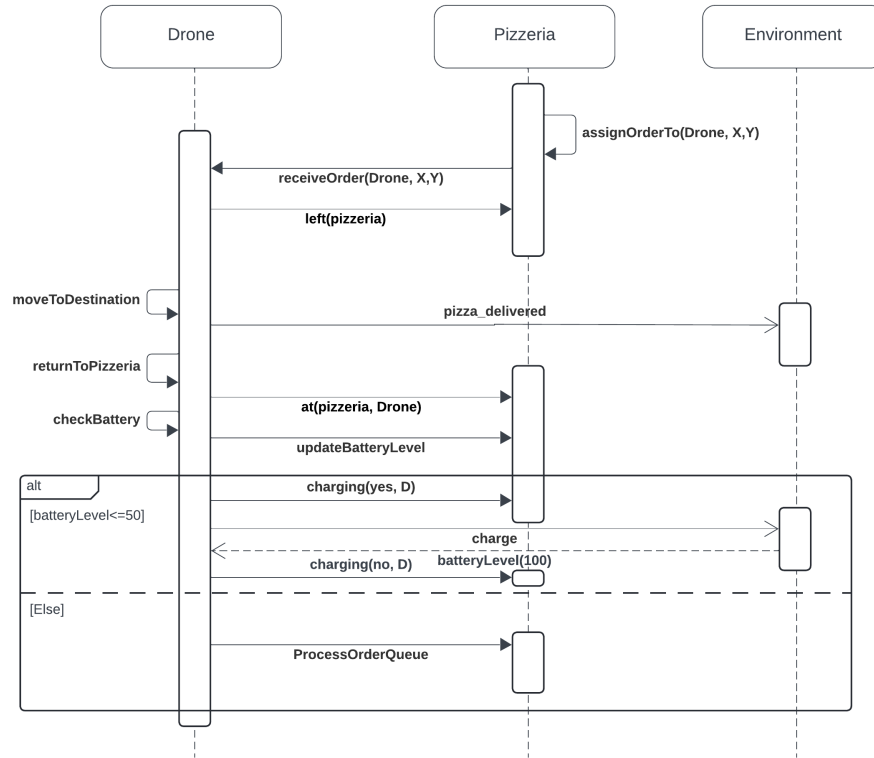


Figure 6: Delivery Sequence Diagram

Delivery Sequence Diagram Figure 6 illustrates the sequence of interactions among the Drone, Pizzeria, and Environment during a pizza delivery. The process starts when the Pizzeria assigns a delivery order to the Drone by sending an `assignOrderTo(Drone, X, Y)` message, where `X` and `Y` represent the delivery coordinates. After receiving the order (`receiveOrder(Drone, X, Y)`), the drone leaves the pizzeria (`left(pizzeria)`) and begins its journey towards the destination (`moveToDestination`).

Once the Drone arrives at the destination, it completes the delivery and notifies the Environment of the successful `pizza_delivered` event. After completing the delivery, the drone starts its return journey to the pizzeria (`returnToPizzeria`).

When the drone arrives back at the pizzeria, it updates its status to indicate that it is at the pizzeria (`at(pizzeria, Drone)`) and checks its battery level (`checkBattery`). The `Drone` then sends an `updateBatteryLevel` message to the `Pizzeria` to inform it of its current battery status.

At this point the sequence includes an **alternative** fragment to handle alternative conditions based on the battery level of the drone. If the battery level is less than or equal to 50%, the drone enters a charging state (`charging(yes, D)`). During this charging phase, the drone communicates with the `Environment` to start the charging process (`charge`) until it reaches full capacity (`batteryLevel(100)`). Once fully charged, the drone exits the charging state (`charging(no, D)`).

If the battery level is above 50%, the drone does not enter the charging state. In either case, once the battery check is complete, the drone notifies the `Pizzeria` to continue processing the next order in the queue (`ProcessOrderQueue`). This sequence ensures that the drone is always ready for the next delivery with an adequate battery level, optimizing the delivery process.

Recovery Sequence Diagram Figure 7 illustrates the sequence of interactions among the `Drone`, `Robot`, `Environment`, and `Pizzeria` during the recovery of a broken drone. The process begins when the `Drone` detects a fault and sends a `brokenDrone(D, CurrentX, CurrentY)` message to notify the `Robot` of its position and status. This message initiates the recovery sequence.

The diagram uses an **alternative** fragment to represent alternative conditions based on the robot's current status. If the `Robot` is not busy, it proceeds with the recovery by invoking the `handleBrokenDrone(D, X, Y)` operation. This starts a series of actions in which the `Robot` takes responsibility for handling the broken drone. The `Robot` first performs the `repairDrone` action to begin the repair process.

Once the repair is complete, the `Robot` updates the `Environment` with new beliefs about the drone's status, including resetting its battery level to 100, marking the drone as not broken (`broken(D, no)`), and setting its position to the coordinates of the pizzeria (`setPosition(26, 26)`). This indicates that the drone has been successfully recovered and is back at the pizzeria.

The `Robot` then sends an `updateBatteryLevel`, `at(pizzeria)`, and `charging(no, D)` message to ensure the `Pizzeria` is aware of the drone's updated state. These updates mean that the drone's successful recovery and repair are complete.

If the `Robot` is already busy when it receives the `brokenDrone` notification, the process follows the **Else** section, where the `Robot` adds the broken drone to its beliefs. This addition ensures that the robot knows the drone's broken status and location, allowing it to handle the recovery after the current process is in execution.

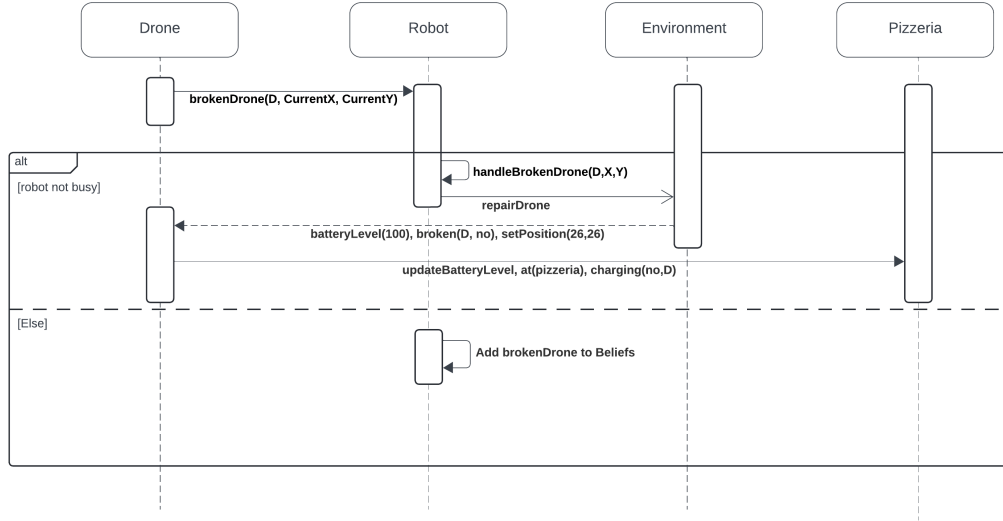


Figure 7: Recovery Sequence Diagram

5 Implementation Details

In this section are presented some non-trivial aspects of the implementation of the project.

5.1 Orders Queue Management

The pizzeria agent is responsible for managing and processing the queue of delivery orders. The processing of the order is fundamental to improve the efficiency of the assignment as shown in 8. We divide the management in more steps:

1. **Queue Insertion and processing:** Generated orders are added at the end of the queue. The pizzeria then processes each order from the head of the queue.
2. **Re-enqueue of not processable orders:** If no drone is available, the order is reinserted at the front of the queue using the plan `!enqueueOrderAtFront`. This mechanism ensures that priority orders remain at the top until a drone becomes free.
3. **Processing remaining orders:** If the maximum number of orders is reached, the agent begins to progressively empty the remaining queue until all orders are processed.

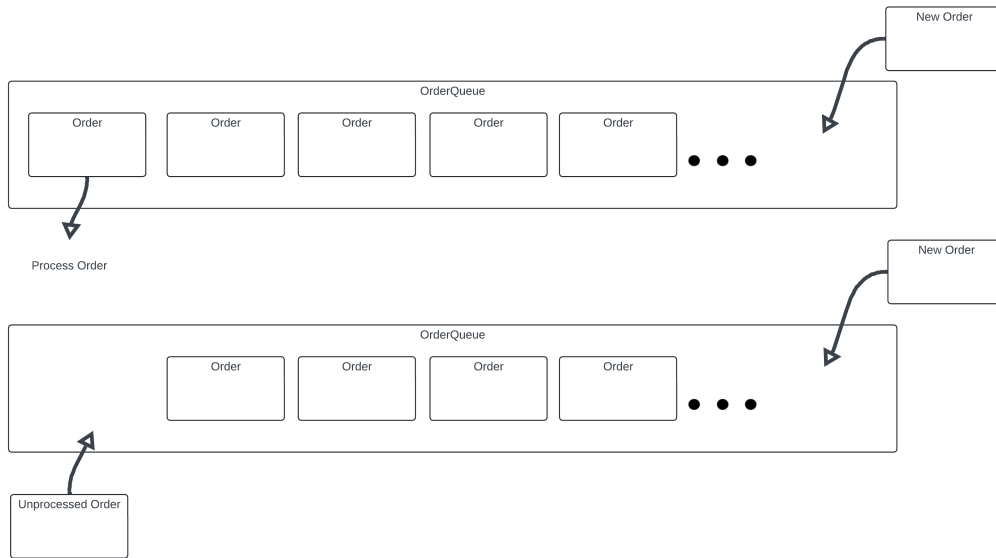


Figure 8: Order Queue Implementation

Listing 1: Orders queue management by pizzeria agent

```

1  +!processOrderQueue <-
2    if (not orderQueue([])) {
3      .print("Process Order Queue");
4      !dequeueOrder(order(X, Y));
5      !checkAvailableDrone(X, Y);
6
7    } else {
8      .print("No order in queue, FINISH!");
9    }.
10
11 +!processRemainingOrders <-
12   if (not orderQueue([])) {
13     .print("process remaining orders...");
14     !dequeueOrder(order(X, Y));
15     !checkAvailableDrone(X, Y);
16     .wait(1000);
17     !processRemainingOrders;
18   } else {
19     .print("Queue is empty");
20   }.
21
22 +!enqueueOrder(order(X, Y)) <-
23   -orderQueue(CurrentQueue);
24   !appendOrder(order(X, Y), CurrentQueue, NewQueue);
25   +orderQueue(NewQueue);
26   .print("Order added to queue: ", NewQueue).
27
28 +!enqueueOrderAtFront(order(X, Y)) <-
29   -orderQueue(CurrentQueue);
30   NewQueue = [order(X, Y) | CurrentQueue];

```

```

31     +orderQueue(NewQueue);
32     .print("Order reinsertion at the front: ", NewQueue).
33
34 +!appendOrder(Order, [], [Order]).
35
36 +!appendOrder(Order, [Head | Tail], [Head | NewTail]) <-
37     !appendOrder(Order, Tail, NewTail).
38
39 +!dequeueOrder(Order) <-
40     -orderQueue([Order | RestQueue]);
41     +orderQueue(RestQueue).
42
43 +!assignOrderTo(Drone, X,Y) <-
44     .print("ORDER ",X, " ", Y," ASSIGNED TO ", Drone);
45     .send(Drone, achieve, receiveOrder(Drone, X,Y)).

```

5.2 Navigation Manager

The `NavigationManager` class is responsible for managing the movement of drones within the environment. This class includes various functionalities that facilitate navigation, taking into account obstacles and occupied locations on the grid.

The constructor of `NavigationManager` initializes the grid size, obstacle locations, occupied positions, and references to the `FlyPizzaModel` for retrieving and updating drone positions.

The `moveTowards` method directs a drone towards its destination by first moving it step-by-step to the destination. If an obstacle blocks the path, the drone attempts to reroute by invoking `moveAroundTheObstacle`, which applies a random directional approach to navigate around obstacles. The final position is updated if it is unoccupied.

The `moveIfNotOccupied` method checks if the intended position is free from other agents or obstacles. This method also synchronizes position updates to prevent conflicts with other agents. The `isAllowedPosition` function permits certain restricted positions, such as the pizzeria location.

The movement logic is implemented in the `moveTowardsDestination` method, which adjusts the drone's coordinates step-by-step towards the destination. When necessary, the `validateCoords` function ensures that the new coordinates remain within grid boundaries.

This class thus provides efficient pathfinding while avoiding conflicts between drones, improving the reliability of the drone's navigation in a dynamic, obstacle-filled environment.

5.3 Failure Simulation & Multithreading

The failure simulation is managed by the environment. After a set interval, the environment randomly triggers a failure for a drone. This functionality is implemented in the `DroneHandler` class, which is responsible for monitoring and updating the operational status of each drone.

- **Class Structure:** The `DroneHandler` class is located in the `env.model.behavior`

package and operates as a runnable task. It receives the drone's identifier, the environment (FlyPizzaEnv), and the model (FlyPizzaModel) with a logger for event tracking.

- **Scheduled Execution:** The class uses a `ScheduledExecutorService` to check the status of each drone at regular intervals, defined by `failureCheckInterval`. The drone status updates every 400 milliseconds, while the failure simulation is triggered with a random delay between 10 and 15 seconds.
- **Random failure generation:** Failures are generated based on a random probability, where a failure event occurs if a randomly generated number is less than 0.25, or 25% probability. The `simulateRandomFailure` method checks if the drone is currently operational (not broken). If the drone is operational, it will be set to a broken state and logged accordingly.

Listing 2: DroneHandler as Runnable

```

1 package env.model.behavior;
2
3 import env.FlyPizzaEnv;
4 import env.model.FlyPizzaModel;
5
6 import java.util.Objects;
7 import java.util.Random;
8 import java.util.concurrent.Executors;
9 import java.util.concurrent.ScheduledExecutorService;
10 import java.util.concurrent.TimeUnit;
11 import java.util.logging.Logger;
12
13 public class DroneHandler implements Runnable {
14     private final String droneName;
15     private final int droneId;
16     private final FlyPizzaEnv environment;
17     private final FlyPizzaModel model;
18     private final Logger logger;
19     private final Random random;
20     private final int failureCheckInterval;
21     private final ScheduledExecutorService scheduler;
22
23     public DroneHandler(String droneName, int droneId, FlyPizzaEnv environment,
24         FlyPizzaModel model, Logger logger) {
25         this.droneName = droneName;
26         this.droneId = droneId;
27         this.environment = environment;
28         this.model = model;
29         this.logger = logger;
30         this.random = new Random();
31         this.failureCheckInterval = 10000 + random.nextInt(5000);
32         this.scheduler = Executors.newScheduledThreadPool(1); //scheduler
33     }
34
35     @Override
36     public void run() {
37         scheduler.scheduleAtFixedRate(() -> environment.updatePercepts(droneName,
38             droneId), 0, 400, TimeUnit.MILLISECONDS);
39         scheduler.scheduleWithFixedDelay(this::simulateRandomFailure,
40             failureCheckInterval, failureCheckInterval, TimeUnit.MILLISECONDS);
41     }

```

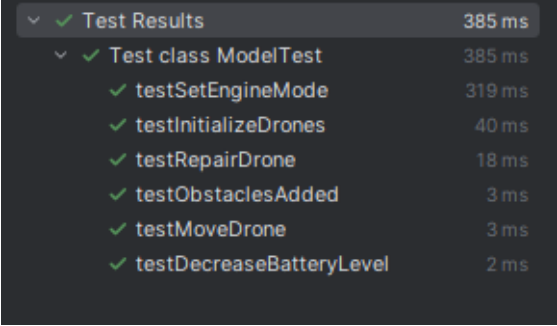
```

38     }
39
40
41     private void simulateRandomFailure() {
42         //generate random number between 0 and 1
43         double randomNumber = random.nextDouble();
44         if (randomNumber < 0.25) { //0.25 corresponds to 25% probability
45             if (Objects.equals(model.isDroneBroken(droneName), "no")) { //if the
46                 drone is not broken I broke it
47                 model.setDroneBroken(droneName, "yes");
48                 logger.info(droneName + " is BROKEN");
49             }
50         }
51     }

```

6 Self-assessment / Validation

Regarding testing, in addition to manual tests, automated tests have been developed using the JUnit framework to verify the model. Within the project, there is a module called "test" containing classes for testing the model.

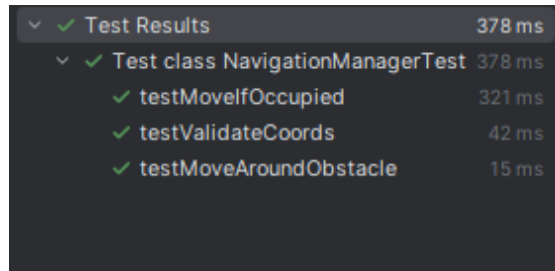


✓ Test Results	385 ms
✓ Test class ModelTest	385 ms
✓ testSetEngineMode	319 ms
✓ testInitializeDrones	40 ms
✓ testRepairDrone	18 ms
✓ testObstaclesAdded	3 ms
✓ testMoveDrone	3 ms
✓ testDecreaseBatteryLevel	2 ms

Figure 9: Model Test

In Figure 9, the main models have been tested to verify that the system logic functions correctly. In the FlyPizza system, agents use the Jason framework: to update perceptions from the environment, an agent calls specific actions that reference the model to execute. In these tests, the main aspects of the model are verified.

In Figure 10, the NavigationManager class has been tested to verify that the drone's movement is correctly defined. This is a complex class: the test checks the coordinates, the movement around obstacles, and cases where the drone attempts to move into an occupied cell.



✓ Test Results	378 ms
✓ Test class NavigationManagerTest	378 ms
✓ testMovelfOccupied	321 ms
✓ testValidateCoords	42 ms
✓ testMoveAroundObstacle	15 ms

Figure 10: Navigation Manager Test

7 Deployment Instructions

Here is the instruction to execute the software.

1. Install JDK on machine.
2. Clone the repository from Github:

```
git clone https://github.com/davidedimarco00/FlyPizza
```
3. Navigate into the project folder:

```
cd FlyPizza
```
4. Execute the run command with gradle:

```
./gradlew runflypizzamas
```

8 Usage Examples

When the applications starts the application provides the user interface: the grid with the object inside. The object are:

- Pizzeria agent: the red circle.
- Drones agents: the colored circles in purple, green and skyblue.
- Robot agent: the yellow circle.
- Obstacles: the black squares.

In figure 11 is presented the scenario where all drones are operative and busy in delivery phase reaching the destination and come back to the pizzeria.

In figure 12 when a drone randomly break by the environment, the robot agent is deployed and starts from the pizzeria to reach the broken agent. In this case the robot is going to recover the green drone that is broken.

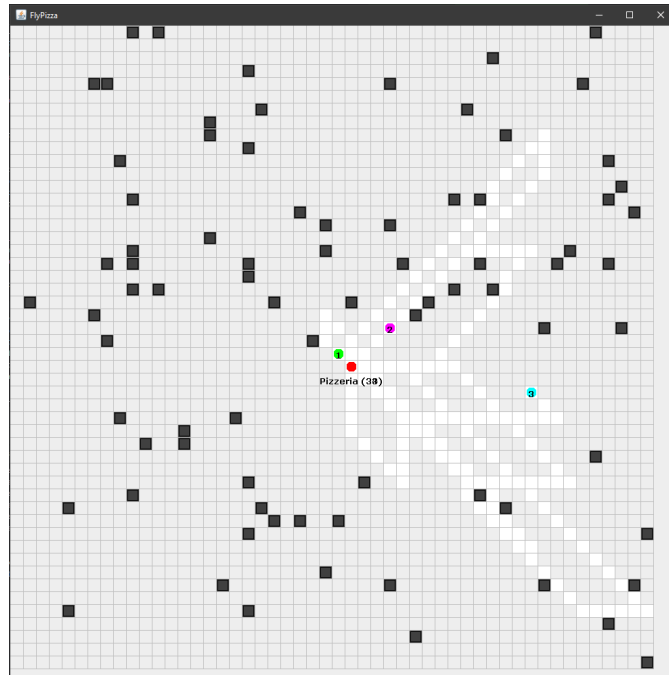


Figure 11: Drone delivery phase

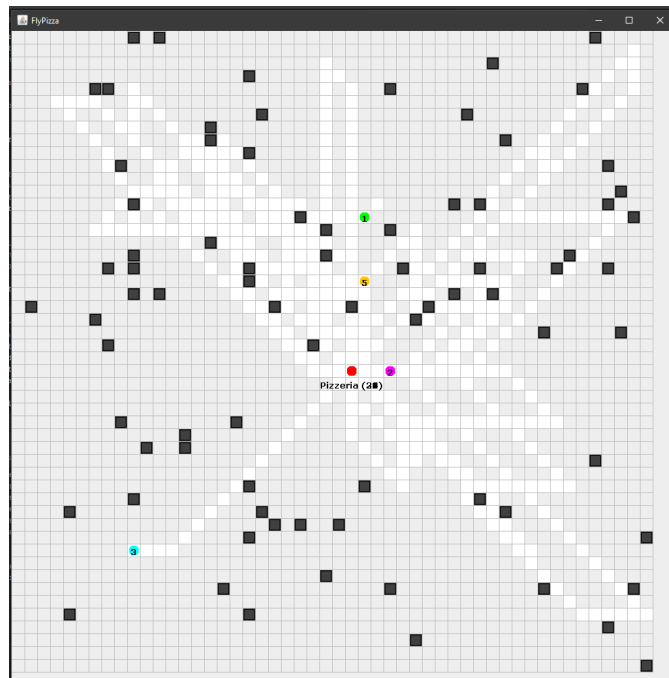


Figure 12: Robot recovery phase

9 Conclusions

This project successfully demonstrates the design and implementation of an autonomous multi-agent drone delivery system for a pizzeria. By the use of the JASON platform, we built a simulated environment where agents (drones, pizzeria, and rescue robot) work together. The use of the BDI (Belief-Desire-Intention) architecture allowed us to develop intelligent agents capable of making autonomous decisions, adapting to environmental changes, and handling failures, all within a structured Multi-Agent System (MAS).

We validated the functionality of the model using JUnit test. The FlyPizza simulation environment provides a foundation for experimenting with drone-based delivery, offering a point of view into how autonomous agents can dynamically adapt to their conditions and terminate a job autonomously.

9.1 Future Works

This project offers several ideas for future development. Some possible directions are:

- Create an agent responsible for receiving and prioritizing orders, simulating a person working in the pizzeria.
- Currently, drones deliver orders without considering destination zones. Assigning each drone a specific zone on the map could help analyze and compare the efficiency of different approaches and algorithms.
- Implement a caller agent to simulate a customer providing order details.
- Enable drones to carry multiple pizzas in a single delivery, making the system more efficient and realistic.

9.2 What did we learned

Working on FlyPizza provided insight into multi-agent system design, particularly in implementing and managing autonomous agents using the BDI architecture. This project deepened our understanding of MAS frameworks and Java-JASON integration. Additionally, we explored practical challenges in synchronization, obstacle avoidance, real-time decision-making and performance within an agent environment. We also learn how to program agent entities in a quite different way from the tradition programming. Finally, we understand how reasonable agent works and how to properly manage error in real time environment.

References

- [1] Bdi architecture. <https://jason-lang.github.io/doc/tutorials/hello-bdi/readme.html>.
- [2] Jason framework website. <https://jason-lang.github.io/>.

- [3] Java website. <https://www.oracle.com/java/technologies/downloads/>.
- [4] Junit. <https://junit.org/junit5/>.