Intelligent Dungeon Crawler: A Multi-Agent System with Chatbot Assistance

Andrea Stucchi
s4942270@studenti.unige.it

February 12, 2025

1 Details of the proposal

1.1 Full specification of the proposal

The project consists of a dungeon crawler application in which the user must conquer a fixed number of rooms presented in sequential order. Access to the next room is unlocked after successfully completing the current one. A room is considered complete when all enemies within it are defeated. In addition, rooms can contain "power-ups," which are objects designed to enhance user capabilities, such as increasing health, defense, or attack power. The final room of the dungeon will feature a "boss," a significantly stronger enemy compared to the standard ones. Defeating the boss signifies the completion of the dungeon. In addition to this, a chatbot will always be available to provide interactive assistance to the user.

1.1.1 Details

The user is provided with fixed attributes for health, attack power, and defense. Two types of attacks are available, a melee attack for close-range combat and a ranged attack for long-distance combat. It can navigate freely within rooms. All actions are assigned to specific keyboard keys.

Enemies share the same fundamental attributes as the user (health, attack power, and defense), but execute their actions automatically. The boss is an enhanced version of standard enemies, featuring boosted attributes (e.g., higher health, defense, and attack power) and a unique special attack.

Enemies will be agents: for the time being, there will be four instances for standard enemies and one instance for the boss.

Power-ups provide specific boosts to one of the following attributes: health, defense, attack power.

The chatbot will be just a dummy demonstrative chatbot to show that the integration between Jade/Java and Rasa works

1.1.2 Agents Communication Patterns

Player Spotted

1. Trigger: An enemy detects the player within close range.
2. Action: The enemy sends an alert to all other enemies in the room.
3. Response: All alerted enemies move to the player's last known position

Requesting Reinforcements

1. Trigger: An enemy actively engaged with the player detects that other enemies in the room have not joined the fight.
2. Action: The engaged enemy sends help requests at regular intervals until support arrives or the fight ends.
3. Response: Idle enemies receive the request and move to join the battle.

Boss Alert

1. Trigger: All enemies in the room fail to defeat the player within a specific time limit.
2. Action: The enemies inform the boss about the situation.
3. Response: The boss enhances its attributes (e.g., health, defense, and/or attack) and prepares for the encounter.

Low Health Escape

1. Trigger: An enemy's health drops below 30%.
2. Action: The enemy communicates its intention to retreat to all other enemies in the room and seeks a power-up location, if available.
3. Response: Nearby enemies position themselves between the retreating enemy and the player to cover its escape. The retreating enemy moves toward the nearest power-up.

Power-Up Notification

1. Trigger: The player collects a power-up.
2. Action: Nearby enemies are notified (because they hear the collection), who inform all room-based enemies about the event
3. Response: Enemies that have not engaged the player will move toward the power-up location.

1.1.3 Implementation

The project will primarily be implemented in JADE, except for the chatbot which will be developed in Rasa. Consequently, an integration between Rasa and JADE will need to be developed. The primary focus of the NLP part of this project will be the development of this integration. Hence, for the initial phase, the chatbot will remain minimal and primarily serve as a demonstration of the functionality of the Rasa-JADE integration.

1.1.4 Possible Evolutions

- Make the chatbot more sophisticated
- Increased Complexity:
  - Adding more rooms to the dungeon to create longer and more challenging adventures.
  - Introducing multiple dungeons with varying themes, layouts, and difficulty levels.
- Enhanced Enemy Variability:
  - Expanding the variety of enemies, including unique abilities or characteristics.
  - Introducing new enemy types to diversify combat scenarios.
- Expanded Power-Ups and Equipment:
  - Adding more types of power-ups to provide greater strategic depth.
  - Introducing different objects and equipment that the user can collect and use.
  - Implementing an inventory system to allow the user to store and manage items.

- Character Progression:
  - Introducing an experience system where the user gains experience points (XP) by defeating enemies.
  - Allowing the user to level up, enhancing attributes such as health, attack power, or defense.
- In-Game Economy:
  - Incorporating a currency system that rewards the user for dungeon exploration and defeating enemies.
  - Using the currency to enable purchasing items or upgrades.
- Hub Area:
  - Adding a neutral zone or hub where the user can:
    - Rest and recover health.
    - Interact with NPCs or a shop to spend currency.
    - Acquire new items, upgrades, or additional power-ups.
- VR rooms
- Turn-based gameplay (e..g. controlling the player via chatbot):
  - In a turn-based version of the game, the chatbot will be used to control a JADE agent (e.g. if the player wants to move north, ask the chatbot "move north" and so on). This approach becomes particularly engaging when the chatbot enforces game rules and constraints. If the player asks the chatbot to perform an action that is not possible (e.g. using an attack they don't possess or attempting to move north when blocked by an obstacle) the chatbot won't allow the user to do so. This interaction creates an interesting dynamic between the chatbot, the multi- agent system (MAS), and the user. The chatbot ensures that only valid actions are executed, providing immediate feedback and maintaining consistency with the game's mechanics.

1.2 The kind of your proposal

Creative

1.3 The range of points/difficulty of your proposal

Hard

2 Introduction

This project's objective is to create a Java Dungeon Crawler application. To win, the player has to conquer three rooms. Four "standard" adversaries and a boss are included in the program; they are all JADE agents. In order to defeat the player, these agents are made to work together and coordinate as necessary. according to 1.1.2. The power-ups in the first two rooms improve the health, defense, or attack power of the entity that gathers them.

The player can attack with melee or ranged attacks, but enemies can only attack with melee, except for the boss. The boss has a special attack that has a 20% chance of being triggered, dealing double the damage of a normal one. All "living" entities in the game have health points, a defense power, and an attack power. These attributes dictate the results of combat interactions.

Furthermore, the user can communicate with an integrated chatbot created in Rasa at any point during the dungeon experience. The integration acts as a proof of concept for the successful bridge between Rasa and Java/Jade, even though the chatbot is mostly demonstrative. This integration demonstrates how multi-agent systems and natural language processing can be combined for interactive applications like gaming.

Because the context naturally demands agents (enemies) to communicate, interact, collaborate, exchange information, and work autonomously in order to accomplish their shared goal of defeating the player, the decision to create this application as a multi-agent system was well-justified. This method improves the gaming environment's complexity and realism while simultaneously adhering to the principles of multi-agent systems.

2.1 Initial impact and Instructions

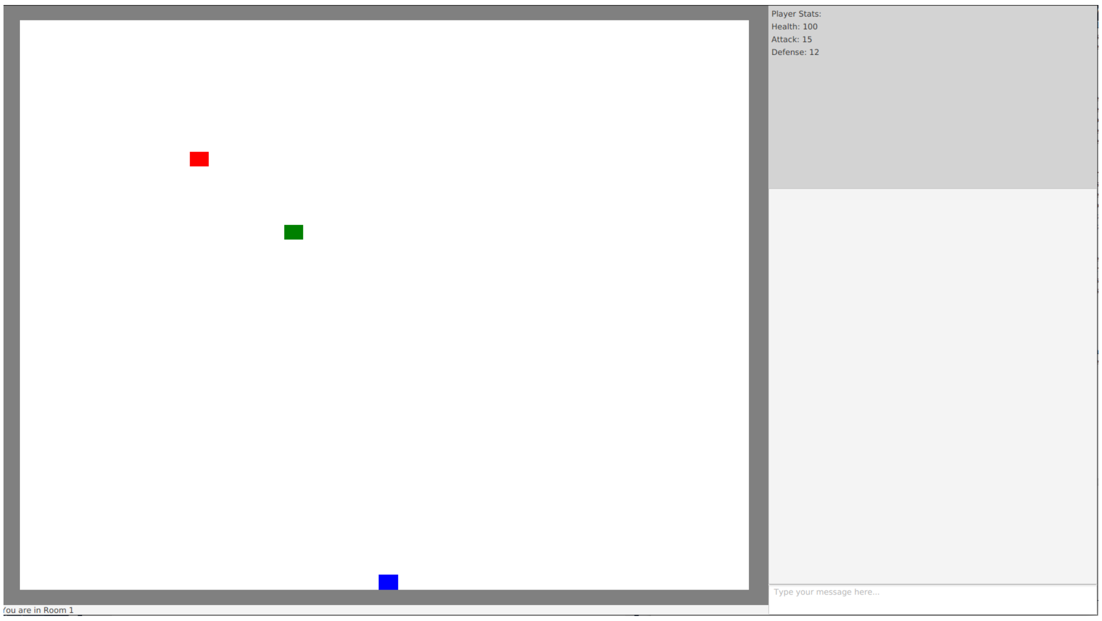
The program's simple GUI (Figure 1) makes it easy for the user to engage with the game. This interface was created to improve the gameplay experience and provide a visual representation of the agents' actions in real time. In addition to showing the player's current room, the GUI also has real-time updates on the player's attributes (in the upper-right corner) and a chat interface (on the right) for engaging with the built-in chatbot. In addition to showing messages from the agents, this chat is the main channel of contact between the user and the chatbot.

The WASD keys (W for up, A for left, S for down, and D for right) are used by the player to travel throughout the game. Mouse inputs are utilized to control combat: distance attacks are controlled by the right mouse button, while melee attacks are controlled by the left. The player must click on the target in order to assault an enemy. By placing the player close to a power-up and hitting the E key, you can collect them. An input field in the bottom-right corner of the screen allows the player to interact with the chatbot by typing messages to start or carry on discussions.

The tiles are visually organized as follows:

- A **gray** tile represents an impassable **wall**
- A **white** tile represents the walkable **floor**
- A **blue** tile represents the **player**
- A **red** tile represents an **enemy**
- A **green** tile represents a **power-up**

In order to finish the game, the user needs to defeat all the enemies inside a room. Notice that by getting close to an enemy, this will spot the user and engage with them. The player can effortlessly traverse the game, fight, and communicate with the chatbot thanks to this user-friendly control system and the visual input the GUI provides. This makes for a seamless and engaging gaming experience.



Player Stats:
Health: 100
Attack: 15
Defense: 12

ou are in Room 1

Figure 1: The Application GUI at start up

2.1.1 How to run the application

In the project/ folder there are two .sh files.
First of all run from the commnad line the command `run_rasa.sh` to start the rasa bot.
Only when the rasa server is up and running, run the command `run_app.sh` to start the application.
The application needs Rasa installed and was developed on Linux with Java 19.

3 Implementation

The development of this project can be divided into four main parts:

- Core Java Classes: These classes serve as the application's backbone, managing the Dungeon Crawler's overall structure, entity management, and game logic.
- Classes and Behaviours of JADE Agents: These classes implement the agents that stand behind the boss and the enemies. Their actions specify how they communicate and interact with the environment.
- Utility Classes: These classes offer common utilities used throughout the application.
- Rasa: This part involves creating a chatbot that uses Rasa to show very basic and illustrative natural language interactions in the game.

In order for the application to operate as a coherent and dynamic multi-agent system, each component is essential.

3.1 Core Java Classes

3.1.1 Position Class

The Position class represents a 2D position in a coordinate system with x and y coordinates. Provides utility methods for operations such as translation, range checks, distance calculation, and parsing from string.

Its methods are as follows:

- Various getters and setters to interact and manipulate the class attributes as needed.
- `translate(int dx, int dy)`: which returns a new `Position` translated by dx and dy.
- `isInRange(Position other, int range)`: to check if two position are within a given range
- `isAdjacentTo(Position other)`: to check if two positions are adjacent.
- `distanceTo(Position other)`: to compute the Euclidean distance between two positions
- `toString()` and `fromString(String, position)` to manipulate `Position` to and from `String`
- `equals(Position other)`: an override to check equality between different instances of the class `Position`

3.1.2 GameEntity Class

It is a superclass that represents a game entity, encapsulating attributes such as position, room, health, attack, defense, and ranged attack range. It provides shared functionalities for subclasses, including movement, attacking, and damage handling.

Its methods are as follows:

- Various getters and setters to interact and manipulate the class attributes as needed.
- Methods used for manipulate the main attributes of a `GameEntity`: `updateAttribute`, `increaseHealth`, `increaseAttackPwr` and `increaseDefensePwr`.
- Utility checking methods such as: `isMoveValid` (to check if the entity can move to the specified position), `isAlive`, `isInRangedRange`, `isInMeleeRange`.
- Movements methods to deal with the movements inside a room:
  - `move(Position newPosition)`: Validates whether the move to the specified `newPosition` is valid. If the move is valid, it invokes the `processRoomMove(Position newPosition)` method.
  - `processRoomMove(Position newPosition)`: Handles the actual movement logic within the `currentRoom` of the `GameEntity`, updating the grid cells to reflect the entity's new position.
- Combat methods that deal with attacking an entity and taking damage by another:
  - `attack(GameEntity target)`: Represents a general attack to a `GameEntity`. It consists of performing an attack roll via the `performAttackRoll(GameEntity target)` method to determine whether the attack hits the target. It is also responsible for making the target take damage in case of a successful attack.

  - `performAttackRoll(GameEntity target)`: Simulates the roll of a D20 die (a die with 20 faces). The attack roll is considered successful if it is strictly greater than the target's defense power.
  - Other straightforward methods such as: `meleeAttack`, `takeDamage`, `kill`, and `rangedAttack`. Note that the `rangedAttack` method does nothing, as not every `GameEntity` necessarily has a ranged attack. The `kill` method simply clears the respective tile in the room, as everything else is handled by the other classes.

3.1.3 EntityType Enum

The types of entities in the game are defined using an **enumerable (enum)**, which includes two constants:

- `PLAYER`: Represents the player entity.
- `ENEMY`: Represents all enemy entities.

This enum provides a simple and efficient way to categorize and distinguish between the two primary entity types in the game.

3.1.4 Player Class

This is a subclass of `GameEntity` that represents a **Player** in the game. It inherits common functionalities from `GameEntity`, such as movement, health management, and basic attacks, while extending it with **player-specific logic**. This includes handling **ranged attacks** and managing **ranged ammunition**, which are unique to the player entity.

Its methods are as follows:

- Getter and setter for `rangedAmmo`.
- Getter for the `EntityType`.
- `rangedAttack` method to implement the ranged attack logic, which just consists of diminishing the `rangedAmmo` of the player and calling the `attack(GameEntity target)` superclass method. No further checks are needed since everything is controlled by other classes that use it.

3.1.5 StandardEnemy Class

This is a subclass of `GameEntity` that represents a **StandardEnemy** in the game. It inherits common functionalities from `GameEntity` just like `Player`. It just provides, apart from the constructors, default attributes and support for statistics enhancement.

Its methods are as follows:

- Getter for the `EntityType`.
- `enhanceAttributes()`: empty since a `StandardEnemy` cannot enhance its attributes. It will be handled by the `BossEnemy` class.

3.1.6 BossEnemy Class

This is a subclass of `StandardEnemy` that represents a **BossEnemy** in the game. It inherits common functionalities from `StandardEnemy`, while extending it with **boss-specific logic**. This includes handling **special attack with charges** and proper enhanced attribute-boosting capabilities

Its methods are as follows:

- `enhanceAttributes()`: Calls the methods of `GameEntity` to boost the attributes: `increaseHealth`, `increaseAttackPwr`, `increaseDefensePwr`.
- `specialAttack(GameEntity target)`: Checks if there are enough charges to perform the attack. If so, it reduces the charges and deals damage to the target with double the attack power. The 20% success probability of this attack is handled by another class that uses it.

3.1.7 PowerUpType Enum

The types of power-ups in the game are defined using an **enumerable (enum)**, which includes three constants:

- `HEALTH`: Increases the entity's health.
- `ATTACK`: Increases the entity's attack power.
- `DEFENSE`: Increases the entity's defense.

This enum provides a simple and efficient way to categorize and distinguish between the different power-up types available in the game.

3.1.8 PowerUp Class

The `PowerUp` class represents a power-up in the game. Power-ups increase specific attributes of a `GameEntity` (health, attack, or defense) by a percentage when collected. Each power-up is defined by its type, `boostPercentage`, and position. A default boost percentage is also provided.

Its methods are as follows:

- Getters for `type` and `position`.
- `applyTo(GameEntity collector)` and `applyBoostByType(GameEntity collector)` to implement the logic of applying the specific power-up boost to the `GameEntity` that has just collected it. The boost applied to the collector varies based on the power-up type.

3.1.9 RoomTileType Enum

The types of tiles present in a room are defined using an **enumerable (enum)**, which includes five constants:

- `EMPTY`: Represents an empty walkable tile.
- `WALL`: Represents a wall that blocks movement.
- `PLAYER`: Represents the player.
- `ENEMY`: Represents an enemy.
- `POWERUP`: Represents a power-up.

This enum provides a structured and efficient way to categorize the different tile types in a room.

3.1.10 Room Class

The `Room` class represents a game room with a grid of tiles of a specified width and height, along with enemies and power-ups. It manages the placement, movement, and removal of entities within the room.

Its methods are as follows:

- Various getters to access and manipulate class attributes as needed.
- Methods for initializing the room grid with walls and empty tiles: `initializeGrid` and `isBoundaryTile`.
- Methods for managing entities inside the room: `addEnemy`, `removeEnemy`, `addPowerUp`, `removePowerUp`, `addPlayerToGrid`, and `collectPowerUpAt`. The last one is used to collect a power-up found inside the room at a given position.
- Methods for various grid operations: `clearGridCell`, `updateGridCell`, and `updateGridWithEntity`, which acts as a wrapper for `updateGridCell` by passing the correct `tileType` for a given entity (`Player`, `PowerUp`, `Enemy`).
- Utility methods such as `clearRoom`, `allEnemiesDefeated`, and `canMoveTo`. The last one checks if a given position is a valid, walkable location inside the room.
- Various entity helper methods:
  - `getPositionFromEntity(Object entity)`: Retrieves the position of an entity within the room. The parameter is an `Object` since `Player`, `StandardEnemy` and `PowerUp` are not of the same class.
  - `getTileTypeForEntity(Object entity)`: Returns the corresponding `RoomTileType` for a given entity.

3.1.11 Dungeon Class

The `Dungeon` class represents a dungeon consisting of multiple rooms. It provides methods for managing rooms, retrieving specific rooms, and querying the size and position of rooms within the dungeon.

Its methods are as follows:

- Getter method for retrieving the list of rooms in the dungeon.
- `size()`, to get the size of the dungeon (i.e., the number of rooms).
- `addRoom(Room room)`, to add a `Room` to the dungeon.
- `getRoom(int index)`, to retrieve a `Room` by its index in the dungeon, if present, and `getRoomIndex(Room room)`, to obtain the index of a given `Room`, if present.

3.1.12 RasaBridge Class

The `RasaBridge` class represents the bridge between Java and Rasa. It is mainly characterized by the endpoint of the `RasaBot`. It then defines additional parameters used to communicate with Rasa for different purposes:

- `RASA_BOT_ENDPOINT`: The main Rasa endpoint with which to communicate to have a conversation.

- `DEFAULT_CONVERSATION_ID`: The default conversation ID if the user using the bridge does not want to set it. Note that every conversation with Rasa has a specific conversation ID. If you want to see the "memory" of a conversation, you should refer to the same conversation.
- `conversation_id`: The conversation ID representing the current conversation with Rasa.
- `RASA_TRACKER_EVENTS_ENDPOINT`: The endpoint of the tracker events, in order to send messages that can change the internal state/memory of Rasa (e.g., changing slot values).
- `rasaTrackerEventsURL`: The full URL of the Rasa Tracker Events.

Its methods are as follows:

- `sendMessageToRasaBot(String message)`: Sends the specified message to the Rasa endpoint by opening a connection with it, wrapping the message into a JSON payload (since Rasa only communicates with JSON), and sending it to the endpoint. It then processes the Rasa response by extracting the response message from the received JSON payload.
- `sendMessageToRasaTrackerEvents(Map<String, String> jsonPayloadPropertiesToSend)`: Takes as input a map of `(property_name, property_value)` pairs for the JSON message to send to the Rasa Tracker Event. The Rasa Tracker Event requires messages in the form of JSON payloads that specify at least the type of the event (e.g., slot). In the case of a slot event, it also requires the name of the slot and its (new) value. This method constructs the JSON payload from this map and sends it to the Rasa Tracker Events.
- `setUpConnection(String endpoint)`: Sets up a connection with the given endpoint.
- `buildJsonPayload(Map<String, String> properties)`: Builds a JSON payload from a map of `(property_name, property_value)` pairs.
- `sendMessage(String inputJson, HttpURLConnection connection)`: Sends the given message through the specified connection.
- `getResponsePayload(HttpURLConnection connection)`: Retrieves the response payload from the given connection.
- `extractResponse(String responsePayload)`: Since Rasa sends responses in JSON format, this method extracts the text response from the received JSON payload.

This class provides a clean and easy-to-use way to communicate with a Rasa ChatBot via Java. A potential improvement could be enabling support for multiple conversations within the same `RasaBridge` by keeping track not only of the current conversation ID but also of all active conversation IDs, and allowing the current conversation ID to be dynamically changed. However, this feature was not implemented, as it is not core to the goals of the `DungeonCrawler` (only the user should communicate with the bot, meaning one conversation throughout the game).

3.1.13 GameManager Class

The `GameManager` class orchestrates the game, controlling its main flow, player interactions, agent spawning, rooms, and integration with systems like JADE and Rasa.
It consists of a wide variety of attributes, due to its central role in the application:

- Constants:

  - `RASA_BASE_URL`: The base URL of the associated Rasa endpoint. Used to establish the bridge and communicate with it.
  - `ATTACK_COOLDOWN_MS`: The player melee attack cooldown.
  - `RANGED_ATTACK_COOLDOWN_MS`: The player ranged attack cooldown.

- Core Game Components:
  - `dungeon`: The dungeon containing all the rooms of the game.
  - `player`: A reference to the player.
  - `currentRoom`: A reference to the current room being played.
  - `currentRoomIndex`: The index of the current room being played, for convenience.

- JADE Agent Management (everything related to JADE):
  - `runtime`: The JADE runtime.
  - `container`: The JADE container for managing agents.
  - `proxyAgentController`: A controller for a `ProxyAgent` used to communicate with agents from the `GameManager` class, if needed (enables Object-to-Agent Communication).
  - `bossAgentAID`: The boss agent AID, for convenience.
  - `enemies`: A map that associates the name of an agent with the enemy instance associated with it. This allows for a direct collection of mappings between the two.

- Game Listeners and Synchronization:
  - `gameEventListener`: A listener for game events (the `DungeonCrawler` class, for managing the UI).
  - `guiReadyLatch`: A synchronization latch to check for the GUI's readiness before starting to act.

- Cooldowns and Task Scheduling:
  - `isAttackOnCooldown`.
  - `scheduler`: A scheduler used to manage cooldowns.

- Rasa Integration:
  - `rasaBridge`.

Its methods are as follows:

- Core Game Setup Methods
  - `setupJADE()`: It sets up the JADE runtime and creates a proxy agent for Object-to-Agent Communication.
  - `initializeGame()`: It initializes the game by creating rooms, starting the current room, and initializing the player and enemies, placing them in their positions.
  - `setupRooms()`: An auxiliary method used by `initializeGame()` to create the three rooms of the application, along with their respective enemies and power-ups (if present), and adding them to the dungeon. It is also responsible for spawning the boss agent inside its room (the final one).

  - `createRoom(int width, int height, List<StandardEnemy> enemies, List<PowerUp> powerUps)`: An auxiliary method used by `setupRooms()` to create a room given its dimensions, enemies, and power-ups.

- Player and Room Management
  - `setCurrentRoom(Room room)`: Sets the current room to the given room and adds the player to it.
  - `movePlayerToNextRoom(Player player)`: Logically moves the player to the next room in the dungeon after clearing the current one. It also notifies the `gameEventListener` of the action, so that the GUI can make the proper changes.
  - `playRoom(Room room)`: For each `StandardEnemy` inside the given room, it spawns the corresponding `StandardEnemyAgent`. It also communicates with the `rasaBridge` via `sendMessageToRasaTrackerEvents` to update the information regarding the number of alive enemies for the Rasa ChatBot (to demonstrate its proper functioning).
  - `movePlayer(Position newPosition)`: Moves the player to the given position via the `move` method of `Player`.

- Enemy Management
  - `spawnBossAgent` and `spawnStandardEnemyAgent`: Spawn the respective type of agent using the auxiliary method `spawnAgent`.
  - `spawnAgent(StandardEnemy enemy, String agentClassName, AID bossAgentAID)`: Spawns an agent by creating a new agent inside the JADE container, with a name depending on the class of the agent and the hashcode of the enemy instance, to ensure unique naming. For convenience, an enemy agent is instantiated with the `bossAgentAID` as well.
  - `findNearbyEnemies(Position powerUpPos)`: Given the position of a power-up, this method finds all the enemies within a 10-tile range from the power-up position inside the current room.
  - `notifyNearbyEnemies(Position powerUpPos)`: Given the position of a power-up, this method uses the auxiliary method `findNearbyEnemies` to find nearby enemies and then leverages the `ProxyAgentController` to notify all of them about the power-up collection at the given position, by sending them an `INFORM ACLMessage` with this information.
  - `sendUsingProxyAgent(ACLMessage message)`: Uses the `ProxyAgentController` to send the specified message to the active JADE enemy agents.
  - `removeEnemy(StandardEnemy enemy)`: Removes an enemy from the game. It removes it from the current room, notifies Rasa that there's one less alive enemy (to update its knowledge about alive enemies), and determines if the player can proceed to the next room. If so, it calls the auxiliary method `movePlayerToNextRoom`. It also notifies the `gameEventListener` of the enemy removal, so that the GUI can make the proper changes.

- Combat Methods:
  - `performAttack(boolean isOnCooldown, Supplier<Boolean> rangeCheck,`
    `Supplier<Boolean> attackAction, int cooldownDuration,`
    `String cooldownMessage, String rangeErrorMessage,`
    `String missedErrorMessage)`: Takes as input whether the attack is on cooldown, a function to check if the attacker is within range, a function to perform the attack, the cooldown duration, and the log messages for failure. It executes an attack for a given attacker, ensuring it is within range, not on cooldown, and successfully performed. In case of failure, it logs a message in the chat via the `logMessage` auxiliary method.

- `setCooldown(int cooldownDuration)`: Sets the attack cooldown by scheduling a thread that sets the attack cooldown flag to false after `cooldownDuration` milliseconds.
- `meleeAttack(GameEntity attacker, GameEntity target)`: Leverages the `performAttack` auxiliary method to perform the attack by passing the proper parameters. The functions passed to this call in order to check the range and to perform the attack are the methods `isInMeleeRange` and `meleeAttack` of the `GameEntity` class.
- `specialAttack(BossEnemy attacker)`: Calls the `specialAttack` method of the `BossEnemy` attacker.
- `meleeAttack(GameEntity target)`: The method that will be called from outside this class to perform a melee attack. It just calls `meleeAttack(player, target)`.
- `rangedAttack(GameEntity attacker, GameEntity target)`: Leverages the `performAttack` auxiliary method to perform the attack by passing the proper parameters. The functions passed to this call in order to check the range and to perform the attack are the methods `isInRangedRange` and `rangedAttack` of the `GameEntity` class. It also checks whether the attacker has enough ranged ammo.
- `rangedAttack(GameEntity target)`: The method that will be called from outside this class to perform a ranged attack. It just calls `rangedAttack(player, target)`.

• Bot Management and Logs:

- `notifyEnemiesAlerted(StandardEnemyAgent notifier)`: Notifies the Rasa chatbot that an enemy has spotted the player, so that the chatbot can inform the player of the occurrence via the chat log. A demonstration that the bridge can also be used to pass messages from an agent to the chatbot.
- `logMessage(String message)`: Used for printing a message in the chat by leveraging the `gameEventListener`.
- `chatWithBot(String input)`: The method used for interacting with the Rasa bot by sending the given input and returning the bot's response. It leverages the `rasaBridge` to communicate with the bot.

• `onPowerUpCollected(GameEntity collector, PowerUp powerUp, Position position)`: A method used to deal with the collection of the specified power-up by a `GameEntity`. It applies the power-up boost to the collector and, if the collector is the player, it notifies all nearby enemies using the `notifyNearbyEnemies` auxiliary method.

• Game Over:

- `gameOver()`: Ends the game, cleaning up game resources, shutting down JADE components, and notifying the listener. It uses the following auxiliary methods for cleaning up specific categories of resources.
- `clearGameResources()`: Clears the current room and the enemies' name map.
- `stopProxyAgent()`: Stops the `ProxyAgentController`, ensuring no lingering processes are left running.
- `shutDownJadePlatform()`: Shuts down the JADE container and runtime to release resources allocated by the multi-agent system. Both the JADE platform container and runtime are safely terminated.
- `shutDownScheduler()`: Shuts down the scheduler to terminate any ongoing or scheduled tasks, ensuring no tasks are left active when the game exits.
- `releaseGuiLatch()`: Releases the GUI latch that prevents the interface from hanging, ensuring the game is ready to shut down after releasing the latch.

  - `finalizeGame()`: Performs final clean-up tasks, resetting the player instance and notifying the `gameEventListener` that the game is over, so the GUI can make the proper changes.

- Various getters and setters.

3.1.14 DungeonCrawler Class

The `DungeonCrawler` class is the main entry class for the Dungeon Crawler application. It handles the integration of JavaFX for the game's GUI, manages user input, and bridges the connection between the GUI and game logic (aka the `GameManager`).

It consists of the following attributes, used for managing the UI:

- `gameManager`: the `GameManager` that manages the core game logic and events.
- `currentRoomIndex`: the current room index, for convenience.
- `currentRoom`: the current room the player is in.
- `root`: the main container of the GUI.
- `statsPane`: the pane that will display the player statistics.
- `roomPane`: the pane that will display the current room.
- `logScrollPane`: the scroll pane that will feature the chat for in-game messages.
- `userInputField`: the input field for the player to feed messages.
- `threadPool`: in order to manage background tasks.

Its methods are as follows:

- Main Application Lifecycle
  - `start(Stage primaryStage)`: The main entry point for JavaFX. Sets up the game environment, initializes UI components, and starts the application. It also notifies the game manager when the GUI is ready.
  - `setupGameEnvironment()`: Initializes the game manager, sets the dungeon crawler as the game event listener of the game manager, and sets up the thread pool.
  - `setupUIComponents(Stage primaryStage)`: Configures the JavaFX UI components and initializes the main game window. It prepares the statistics pane, the chat pane, the room pane, and sets up the main events (e.g., player movements upon key pressing).
  - `setupStatsAndLogContainer()`: Initializes the statistics and chat container that appears on the right side of the UI. It separately sets up the statistics pane, the chat pane, the input field, and then wraps them together into the same container.
  - `setupStatsPane()`: Configures the statistics panel to display player health, attack, and defense.
  - `setupLogPane()`: Configures the log pane to display in-game messages.

  - `setupUserInputField()`: Prepares the user input text area and configures behaviour on Enter pressing. In particular, the input field does not have focus when the application starts. It only gains focus once clicked on, and once Enter is pressed (without pressing Shift), the text is consumed, and the input is handled through the `handleUserInput()` auxiliary method.

- User Interaction Handling
  - `handleUserInput()`: Handles the player's chat input by sending it to Rasa via a background thread and adding the user message to the chat log.
  - `addPlayerMessage(String message)`: Appends a message from the player to the chat log by using the auxiliary method `addLogMessage` and simply adding the prefix `"You: "` before the message to make it explicit it's from the player.
  - `addLogMessage(String message)`: Creates a new log message and adds it to the chat thanks to the `addToScrollPane()` auxiliary method, but delegates the graphical operations to the main JavaFX thread.
  - `addToScrollPane(Label logMessage)`: An auxiliary method used by `addLogMessage` to effectively add a new log message to the visual chat log.

- Player Movement and Interaction
  - `setupPlayerMovement(Scene scene, Player player)`: Sets up player movement controls and interaction handling in the game by listening for key press events to trigger movement or interactions. Based on the pressed key, it triggers two different auxiliary methods: one for directional movement and one for interaction.
  - `handleMovementInput(KeyCode keyCode, Player player)`: The auxiliary method used by `setupPlayerMovement` to handle directional movement. If the pressed key is one of W, S, A, D, a new position for the player is computed by translating the current one accordingly, leveraging the `translate` method of the `Position` class. Then it attempts to move the player by using the `movePlayer` method of the game manager. If the player can effectively move to the new position, the movement is performed, and the UI is updated accordingly by delegating the change to the main JavaFX thread.
  - `handlePowerUpInteraction()`: Called when the user presses the `"E"` key. It checks if, in the positions adjacent to the player, there is indeed a power-up to collect. If so, the auxiliary `collectPowerUp()` method is called to effectively collect the power-up, otherwise an "error" message is logged in the chat.
  - `collectPowerUp(GameEntity collector, Position powerupPosition, Room currentRoom)`: Auxiliary method used by `handlePowerUpInteraction` to collect a power-up at a specified position by leveraging the `collectPowerUpAt()` method of the current room. If present, the game manager is notified of the event by calling its `onPowerUpCollected()` method, and the UI is updated as usual.
  - `handleAttack(Position position, boolean isRanged)`: Handles attacking a target at the given position and supports both ranged and melee attacks. It first checks if there's an enemy at the specified position. If so, depending on the type of the attack, the corresponding method of the game manager is called to perform the game logic. If the attack was successful, the target entity is visually highlighted thanks to the `highlightEntityTile()` method. Finally, it checks if the enemy is dead, if so, a log message is shown in the chat, and the corresponding enemy is removed from both the current room and the game manager.
  - `handleRangedAttack(Position position)`: A wrapper method for the `handleAttack()` method that simply calls this one with the specified position and the `isRanged` flag set to true.

  - `handleMeleeAttack(Position position)`: A wrapper method for the `handleAttack()` method that simply calls this one with the specified position and the `isRanged` flag set to false.

- Room and Dungeon State Management
  - `initializeRoomUI(RoomTileType[][] grid)`: Initializes the room UI by generating a grid of labels representing the room tiles. Each label corresponds to a tile in the room and is clickable for dealing with attacking an enemy. For every tile, it handles the clicking events for both the right and left mouse buttons, in order to manage both melee and ranged attacks. It leverages the `handleMeleeAttack()` and `handleRangedAttack()` methods to handle these events.
  - `updateRoomUI(Room room)`: Updates the room UI by creating a new grid if needed with the `initializeRoomUI()` auxiliary method or refreshing the visuals of the existing grid based on the current state of the room using the `updateRoomTile()` auxiliary method.
  - `highlightEntityTile(Position position)`: Highlights a specific tile in the room (e.g., when an entity is attacked) and temporarily changes the tile's style, then resets it after a delay.
  - `updateRoomTile(Position position, Room room)`: Updates a specific tile in the room based on its current state with the `updateRoomTileStyles()` auxiliary method.
  - `updateRoomTileStyles(Label cell, RoomTileType tileType)`: Effectively changes the visual style of the given tile based on its type.
  - `showRoom(int roomIndex)`: Displays the specified room by room index, updating the room UI and refreshing its content. It checks if the room index exceeds the dungeon size in order to trigger the end of the game. Otherwise, it leverages the game manager to gather the information about the room to show and delegates the room visual changes to the `updateRoomUI()` auxiliary method.

- Player Stats and Game Lifecycle
  - `updatePlayerStats()`: Dynamically updates the player's stats in the side statistics panel. It clears the current panel and repopulates it with updated information.
  - `shutDownGame()`: Shuts down the game cleanly. It ends the game session by notifying the game manager and closes the application.
  - `endGame()`: Displays a congratulatory message to the player when the game is ended.
  - `onWin()`: Handles the victory event when the player wins the game by delegating to the generic game over handler with a win condition.
  - `onGameOver(boolean win)`: Effectively handles the game-over event, by clearing the whole GUI and showing an appropriate message to the user based on victory or failure.

- GameEventListener Implementations
  - `onEntityMoved(Position oldPosition, Position newPosition, EntityType entityType)`: Handles the event when an entity moves. Updates only the affected tiles: the old position (cleared) and the new position.
  - `onEnemyKilled(Position position, StandardEnemy enemy)`: Handles the event when an enemy is killed. Updates the tile where the enemy was located to reflect the absence of the enemy.
  - `onPlayerAttacked(Player player)`: Handles the event where the player is attacked by an enemy. Highlights the player's tile briefly, updates the player's stats, and handles the player's death if health reaches 0 by delegating to the game manager with its `gameOver()` method.

  - `onPlayerMovedToNextRoom(Room targetRoom)`: Handles the event where the player moves to the next room. Updates the current room index and reloads the UI to display the new room.
  - `printLogMessage(String message)`: Logs a generic message to the game's log panel.
  - `printPlayerMessage(String message)`: Logs a message specific to the player, prefixing it with `"You:  "` for clarity.

- Main Method
  - `main(String[] args)`: The starting point of the whole application.

3.2 Classes and behaviours of JADE Agents

The approach adopted for the Multi-Agent System (MAS) aims to associate each enemy agent with a specific state, which functions similarly to a finite state machine, reflecting the agent's current behaviour. To facilitate communication, all enemy agents are registered in the JADE directory facilitator along with their corresponding state. This design choice was made because, in the application, communication is based not on the agent's name, but on its current activity. Since an agent's state directly corresponds to its behaviour, this technique streamlines communication, making it easier to identify the recipients of a particular message when needed.

3.2.1 EnemyState Enum

The states an agent can be in are defined using an **enumerable (enum)**, which includes the following constants:

- `IDLE`: Represents the state where the agent is not engaged in any activity.
- `CHASING_PLAYER`: Represents the state where the agent is pursuing the player.
- `CHASING_TARGET`: Represents the state where the agent is chasing a target other than the player.
- `CHASING_POWERUP`: Represents the state where the agent is pursuing a power-up position.
- `REINFORCING`: Represents the state where the agent is giving reinforcements to another enemy.
- `ATTACKING`: Represents the state where the agent is actively engaging in combat.
- `RETREATING`: Represents the state where the agent is retreating from combat.
- `COVERING`: Represents the state where the agent is covering for a `RETREATING` enemy.

This enum provides a clear and organized way to define the possible states for an agent, enabling efficient management of its behaviour and communication.

3.2.2 StandardEnemyAgent Class

The `StandardEnemyAgent` class represents an enemy agent in the game. It is capable of interacting with the game environment, updating its state, and performing specific behaviours like idling, escaping, attacking, and alerting the boss. It defines different constants to deal with the ticking behaviour delays and is mainly characterised by the reference to the `StandardEnemy` of the agent and its current state. For convenience, a reference to the game manager and the boss agent `AID` are stored here.

Its methods are as follows:

- `setup()`: It initializes the agent upon its creation. It prepares the agent by initializing its attributes, registering it in the `Directory Facilitator (DF)`, waiting for the GUI readiness, and initiating its starting behaviours.
- `initializeAgent()`: Auxiliary method used by the `setup` method to extract and initialize the agent parameters from the arguments passed during initialization with the `spawnAgent` method of the game manager.
- `waitForGuiReady()`: It waits for the game interface (GUI) latch to ensure proper setup before the agent runs.
- `initializeBehaviours()`: It initializes and adds the starting behaviours for the enemy agent. It starts with:
  - `IdleBehaviour`: The starting point of an agent.
  - `LowHealthEscapeBehaviour`: An escape behaviour for when the health is low.
  - `WakerBehaviour`: Used to trigger the one-time `BossAlertBehaviour`.
  - An anonymous `TickerBehaviour` to monitor the life of an enemy and terminate it once dead.
- `registerWithDF(EnemyState state)`: It registers the enemy agent in the DF with its AID. It adds a new service as well with its local name and a new property that contains the current state of the agent.
- `updateDFState(EnemyState newState)`: It updates the registration of the enemy agent in the DF with the new state, if it is different from the current one.
- `setupBossAlert(StandardEnemyAgent agent, long delay)`: Auxiliary method used by `initializeBehaviours` to add the `WakerBehaviour` that triggers the `BossAlertBehaviour` after a delay.
- `speak(String message)`: It logs the message into the in-game chat via the game event listener of the game manager.
- `takeDown()`: It cleans up resources and removes the enemy from the game's list of active enemies when this agent is being destroyed.
- Checker methods such as `isBossAgent()` (always false) and `isRetreating()`.
- Various getters.

3.2.3 BossEnemyAgent Class

The `BossEnemyAgent` class represents a boss agent in the game. It is an extension of the `StandardEnemyAgent` and differs from it mainly in its starting behaviours.

Its methods are as follows:

- `setup()`: It just calls the superclass's analogous method.
- `initializeAgent()`: It overrides the superclass method mainly in order to initialise the enemy reference to a `BossEnemy` instead of a `StandardEnemy`.
- `initializeBehaviours()`: It initialises and adds the starting behaviours for the boss enemy agent. It starts with:
  - `IdleBehaviour`: the starting point of an agent.
  - `BossResponseBehaviour`: to respond to the `BossAlertBehaviour` of a `StandardEnemyAgent`.
  - An anonymous `TickerBehaviour` to monitor its life and terminate it once dead.
- A checker method `isBossAgent()`: always returns `true`.

3.2.4 ProxyAgent Class

The `ProxyAgent` class is a utility JADE agent responsible for communicating on behalf of the `GameManager` by enabling Object-to-Agent (O2A) messaging and forwarding messages to other agents.

Its methods are as follows:

- `setup()`: It enables the Object-to-Agent communication and adds an anonymous `CyclicBehaviour()` to process incoming O2A messages.
- `takeDown()`: It disables the O2A communication.

3.2.5 IdleBehaviour Class

The `IdleBehaviour` class manages the idle state of an enemy agent. While in the idle state, the agent monitors environmental triggers, such as the presence of a player or messages from other agents, essentially any trigger that prompts it to take action. Because `TickerBehaviour` allows for efficient, periodic monitoring of environmental triggers, like player presence or inter-agent messages, without the overhead of continuous polling (`CyclicBehaviour`) or the rigidity of a single execution (`OneShotBehaviour`), it is the perfect choice for an enemy agent's idle state. It strikes a balance between responsiveness and resource efficiency by checking for triggers at certain intervals, which is consistent with sentinels and guards' real-world idle behaviours. This method is the most practical option for handling idle situations in JADE since it guarantees scalability, simplicity, and adaptability in dynamic environments.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour, and it uses a `MessageHandler` to handle sending messages and processing responses.

Its methods are as follows:

- `IdleBehaviour(StandardEnemyAgent agent, long delay)`: The constructor, which, apart from initializing attributes, registers the handlers for various incoming messages.
- `onTick()`: The core of the behaviour. It first updates the current state of the agent in the DF. It then checks whether the player is within a 5-tile range; if so, the detection is logged, the game manager is notified (in order to send a message to Rasa), a broadcast message of type `PLAYER_SPOTTED` is sent via the `MessageHandler` to all the `IDLE` and `CHASING_TARGET` enemies with the player's current position and an `INFORM` performative so that the receivers can decide to come to the position to look for the enemy. A new `ChasingPlayerBehaviour` is added to start chasing the player, and finally, the `IdleBehaviour` is removed. It then checks for incoming messages and handles them appropriately via the `MessageHandler`. The behaviour stops if the player dies.
- `registerMessageHandlers()`: It registers all the message handlers for this behaviour, referring to the type of messages an enemy in this behaviour can process. Each message handler maps to a specific event type. An idle enemy can:
  - Respond to an enemy informing it of the player being spotted (`PLAYER_SPOTTED`).
  - Respond to the game system informing it that the player has collected a power-up nearby (`POWER_UP_COLLECTED`).
  - Respond to an enemy informing it that the player has collected a power-up (`POWER_UP_SPOTTED`).
  - Respond to an enemy asking for reinforcements (`REINFORCEMENT_REQUEST`).
  - Respond to an enemy asking for help while retreating (`RETREATING`).

- `processResponse(EnemyState newState, ACLMessage message, String posString)`: It essentially performs the transition from `IdleBehaviour` to `GoingToTargetBehaviour` to make the agent go to the given `posString` position in the game.
- `handlePlayerSpotted(ACLMessage message)`: It handles the `PLAYER_SPOTTED` message by transitioning to the `CHASING_TARGET` state and the `GoingToTargetBehaviour` via the `processResponse` utility method and logs an appropriate message in the chat via the `speak` method.
- `handlePowerUpCollected(ACLMessage message)`: It handles the `POWER_UP_COLLECTED` message by transitioning to the `CHASING_TARGET` state and the `GoingToTargetBehaviour` via the `processResponse` utility method, logs an appropriate message in the chat via the `speak` method, and broadcasts an alert to other agents as well.
- `handlePowerUpSpotted(ACLMessage message)`: It handles the `POWER_UP_SPOTTED` message by transitioning to the `CHASING_POWER_UP` state and the `GoingToTargetBehaviour` via the `processResponse` utility method and logs an appropriate message in the chat via the `speak` method.
- `handleReinforcementRequest(ACLMessage message)`: It handles the `REINFORCEMENT_REQUEST` message by transitioning to the `REINFORCING` state and the `GoingToTargetBehaviour` via the `processResponse` utility method, logs an appropriate message in the chat via the `speak` method, and sends a reply to the sender to acknowledge the request.
- `handleRetreating(ACLMessage message)`: It handles the `RETREATING` message by transitioning to the `CoverRetreatBehaviour`, acknowledging to the ally sender that it is going to cover him, and logs an appropriate message in the game chat via the `speak` method.

3.2.6 GoingToTargetBehaviour Class

The `GoingToTargetBehaviour` class manages the movement of an enemy agent toward a specified target position. While pursuing its target, the agent monitors environmental triggers like the presence of a player or messages from other agents. It is a `TickerBehaviour` for the same reason as in 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour, the current state of the agent, the target position to reach, and it uses a `MessageHandler` to handle sending messages and processing responses.

Its methods are as follows:

- `GoingToTargetBehaviour(StandardEnemyAgent agent, long delay, EnemyState currentState, Position targetPosition)`: Similar to the constructor in 3.2.5.
- `onTick()`: The core of the behaviour. It first updates the current state of the agent in the DF. It then checks if the player is adjacent, so as to attack it. In this case, it transitions to the `AttackingBehaviour` via the `attackIfPlayerIsNear` utility method of the `BehaviourUtils` utility class. It then processes incoming messages if available, and if the agent is not a boss. This check is needed because the boss can indeed have this kind of behaviour, but it's not allowed to process any messages except for the one related to the `BossAlertBehaviour`. It then checks if it has reached the target position and transitions to `IdleBehaviour` if so, via the `stopIfTargetPositionReached` utility method of the `BehaviourUtils` utility class. Finally, it checks if the player is within a five-tile range to start chasing it if present by transitioning to the `ChasingPlayerBehaviour` via the `chaseIfPlayerIsInRange` utility method of the `BehaviourUtils` utility class. It then broadcasts an alert to all the `IDLE` and `CHASING_TARGET` enemies, as in the `onTick()`

of the `IdleBehaviour`. If the enemy does not change behaviour and is still pursuing the target, the `pursueTarget` method of the `MovementUtils` class is called to continue moving toward the target position. The behaviour stops if the player dies.

- `registerMessageHandlers()`: It registers all the message handlers for this behaviour, referring to the type of messages an enemy in this behaviour can process. Each message handler maps to a specific event type. A `GoingToTarget` enemy can:
  - Respond to the game system informing it that the player has collected a power-up nearby (`POWER_UP_COLLECTED`).
  - Respond to an enemy informing it that the player has collected a power-up (`POWER_UP_SPOTTED`).
  - Respond to an enemy asking for reinforcements (`REINFORCEMENT_REQUEST`).
  - Respond to an enemy asking for help while retreating (`RETREATING`).
- `processTarget(EnemyState newState, ACLMessage message, String posString)`: It essentially performs the change of state of the agent and updates the target position, since all the messages it can process are related to reaching a new target position. Therefore, the same behaviour is used, with just an updated position.
- `handlePowerUpCollected`, `handlePowerUpSpotted`, `handleReinforcementRequest`, `handleRetreating`: These methods are similar to those in 3.2.5, but if needed, they use the `processTarget` method instead of the `processResponse` one.

3.2.7 ChasingPlayerBehaviour Class

The `ChasingPlayerBehaviour` class manages the behaviour of an enemy when it is actively chasing the player. This behaviour is triggered when the player is detected within a specified range. It continuously monitors the player's position to pursue, transitions to other behaviours if the player is no longer in sight, the enemy is close enough to attack, or as a reaction to other agents' messages. It is a `TickerBehaviour` for the same reason as in 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour, the current state of the agent, and it uses a `MessageHandler` to handle sending messages and processing responses.

Its methods are as follows:

- `ChasingPlayerBehaviour(StandardEnemyAgent agent, long delay, EnemyState currentState)`: Similar to the constructor of the `IdleBehaviour`.
- `onTick()`: The core of the behaviour. It first updates the current state of the agent in the DF. It then checks if the player is adjacent, so as to attack it. In this case, it transitions to the `AttackingBehaviour` via the `attackIfPlayerIsNear` utility method of the `BehaviourUtils` utility class. It then processes incoming messages if available, and if the agent is not a boss, like in 3.2.6. Then, if the player is out of range, it transitions back to `IdleBehaviour`. Otherwise, it pursues via the `chaseIfPlayerIsInRange` utility method. In any case, it logs an appropriate message via the `speak` method. The behaviour stops if the player dies.
- `registerMessageHandlers()`: It registers all the message handlers for this behaviour, referring to the type of messages an enemy in this behaviour can process. Each message handler maps to a specific event type. A `ChasingPlayer` enemy can:
  - Respond to an enemy asking for help while retreating (`RETREATING`).
- `handleRetreating(ACLMessage message)`: Similar to `IdleBehaviour`.

3.2.8 AttackingBehaviour Class

The `AttackingBehaviour` class manages the attack phase of an enemy agent when engaged with the player. The behaviour ensures the enemy continues attacking the player as long as the player is adjacent. It is a `TickerBehaviour` for the same reason as in 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour, the player to attack, and it uses a `MessageHandler` to handle sending messages and processing responses.

Its methods are as follows:

- `onTick()`: The core of the behaviour. It first updates the current state of the agent in the DF, but only if the state is not `COVERING`. This is because a `COVERING` enemy can attack the player if they get close, but its main purpose will still be to cover its ally. Then, an attack is performed via the `attackPlayer` method if the player is still adjacent. Otherwise, it transitions back to `ChasingPlayerBehaviour`. The behaviour stops if the player dies or if the enemy is retreating. It also processes incoming messages if available and if the agent is not a boss, like in the `GoingToTargetBehaviour`.
- `attackPlayer()`: It executes the attack logic. If the agent is a `BossEnemyAgent`, the additional logic for special attacks is handled. It uses the `specialAttack` and `meleeAttack` methods of the game manager and notifies the game manager's game event listener of the player being attacked, if the attack was successful, in order to update the GUI. In any case, an appropriate message is shown in the chat log via the `speak` method.
- `registerMessageHandlers()`: It registers all the message handlers for this behaviour, referring to the type of messages an enemy in this behaviour can process. Each message handler maps to a specific event type. An `Attacking` enemy can:
  - Respond to an enemy asking for help while retreating (`RETREATING`).
- `handleRetreating(ACLMessage message)`: Like in 3.2.5.

3.2.9 RequestReinforcementsBehaviour Class

The `RequestReinforcementsBehaviour` class manages the behaviour of an enemy agent when reinforcements are required during an active engagement with the player. The agent periodically broadcasts a reinforcement request to other non-engaged agents while it remains in the attacking state. It is a `TickerBehaviour` for the same reason as in 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour and the player to attack.

Its methods are as follows:

- `onTick()`: The core of the behaviour. It just broadcasts a `REINFORCEMENT_REQUEST` to all agents that are `IDLE`, `CHASING_TARGET`, `CHASING_POWERUP`, or `REINFORCING`, by passing the player's position and using a `REQUEST` performative. Semantically speaking, this message requires an action from the other agents; it can be seen as a command from the current agent to instruct other agents to come to its aid. This behaviour is removed if the player dies or the enemy is no longer attacking.

3.2.10 LowHealthEscapeBehaviour Class

The `LowHealthEscapeBehaviour` class handles the behaviour of an enemy agent when its health drops below a critical threshold. The agent transitions into a retreating state and searches for a nearby power-up to recover strength. It is a `TickerBehaviour` for the same reason as 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour.

Its methods are as follows:

- `onTick()`: the core of the behaviour. When the enemy's health drops below 30% and it's not already in a `RETREATING` state, the current state of the agent is updated to the DF. It broadcasts a `RETREATING` message with `REQUEST` performative to all agents that are not `COVERING` to come and cover it while it escapes. It then looks for the nearest power-up position to escape to via the `findNearestPowerUp` auxiliary method. It then logs an appropriate message via the `speak` method and transitions to the `RetreatingBehaviour`. This behaviour is removed if the player dies.
- `findNearestPowerUp()`: it searches for the nearest power-up position in the current room. It prioritizes active power-ups but falls back to any power-up position if none are active, so that the enemy always has a position to escape to.

3.2.11 BossResponseBehaviour Class

The `BossResponseBehaviour` class manages the behaviour of a boss enemy agent when it needs to respond to alerts from its minions. The boss continuously listens for messages and dynamically adjusts its attributes or actions based on those alerts. It is a `CyclicBehaviour`, since it is made to operate indefinitely, making it ideal for situations that call for constant observation and quick reaction times. A `CyclicBehaviour` guarantees that the boss is constantly prepared to process alerts without delay, in contrast to a `TickerBehaviour` that runs at set intervals or a `OneShotBehaviour` that runs only once.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour.

Its methods are as follows:

- `BossResponseBehaviour(BossEnemyAgent agent)`: like the constructor of the 3.2.5.
- `action()`: the core of the behaviour. It just sits and waits for incoming messages and handles them with the appropriate handler once arrived. This behaviour is removed if the player dies.
- `registerMessageHandlers()`: it registers all the message handlers for this behaviour, which refers to the type of messages an enemy in this behaviour can process. Each message handler maps to a specific event type. A `Boss` enemy with this behaviour can:
  - React when it receives an alert from other enemies that the player is still alive (`PLAYER_SURVIVAL_ALERT`).
- `handlePlayerSurvival(ACLMessage message)`: it handles the `PLAYER_SURVIVAL_ALERT` message. Upon receiving the alert, the boss enhances its attributes to prepare for confrontation and removes this behaviour from the boss agent.

3.2.12 BossAlertBehaviour Class

The `BossAlertBehaviour` class handles the alerting of the `BossEnemyAgent` regarding the player's presence. This behaviour is executed as a one-shot action where an enemy informs the boss about the player's survival to enhance its strength. This is why it is a `OneShotBehaviour`.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour.

Its methods are as follows:

- `action()`: the core of the behaviour. It just logs the alert notification and sends a `PLAYER_SURVIVAL_ALERT` message to the `BossEnemyAgent`. This behaviour is removed if the player has died.

3.2.13 RetreatingBehaviour Class

The `RetreatingBehaviour` class manages the behavior of an enemy agent when it is attempting to retreat to a specified target position. This behavior is triggered when the enemy's health is low. The agent moves toward the target position (e.g., a power-up) and transitions out of this behavior upon reaching its destination. It is a `TickerBehaviour` for the same reason of 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour, the current state of the enemy and the target position on which to retreat.

Its methods are as follows:

- `onTick()`: the core of the behaviour. It first of all updates the current state of the agent to the DF and it pursues the target position until reached. Once adjacent to it, the collection of the power-up from the enemy agent is triggered via the `updateDungeon` utility method of the `BehaviourUtils` utility class. Then, a `RETREATING_POWER_UP_COLLECTED` message is sent to all the `COVERING` enemies that the `RETREATING` enemy has successfully retreated and can hence stop covering it. Finally, the retreating enemy transitions to the `IdleBehaviour`. This behaviour is removed if the player has died.

3.2.14 CoverRetreatBehaviour Class

The `CoverRetreatBehaviour` class manages the behaviour of an enemy agent providing cover during the retreat of another allied enemy. The agent moves dynamically to optimal positions and attacks the player to protect the retreating enemy. It is a `TickerBehaviour` for the same reason as 3.2.5.

This behaviour is mainly characterized by the `StandardEnemyAgent` that has the behaviour, the reference to the `StandardEnemy` whose retreating, the position of the player and it uses a `messageHandler` to handle sending messages and processing responses.

Its methods are as follows:

- `CoverRetreatBehaviour(StandardEnemyAgent agent, StandardEnemy retreatingEnemy, Position playerPosition)`: like the constructor of 3.2.5.
- `onTick()`: the core of the behaviour. It first of all updates the current state of the agent to the DF. It then attacks if the player gets adjacent and calculates and moves to the next optimal cover position via the `calculateCoverPosition` and `moveToCoverPosition` utility methods. This behaviour is removed if the player dies, the enemy itself transitions to retreating, the retreating enemy dies or retreats successfully.
- `registerMessageHandlers()`: it registers all the message handlers for this behaviour, which refers to the type of messages an enemy in this behaviour can process. Each message handler maps to a specific event type. A `CoverRetreat` enemy can:
  - Respond to the retreating enemy informing that it has retreated successfully (`RETREATING_POWER_UP_COLLECTED`).
- `handlePowerUpCollected(ACLMessage message)`: it handles the `RETREATING_POWER_UP_COLLECTED` message by transitioning the agent to `IdleBehaviour` as the cover duty is no longer necessary and logging an appropriate message in the chat log.

- `calculateCoverPosition(Position playerPos, Position retreatingPos)`: calculate the optimal cover position for the agent to provide protection between the retreating enemy and the player.
- `moveToCoverPosition(Position coverPosition)`: it moves the agent to the provided cover position. If movement is successful, it notifies the `GameManager` about the position change to make the appropriate GUI changes.

3.3 Utility Classes

In this section, we will briefly take a look at the three utility classes that have been built for this project, regarding behaviour, messages, and movement utils.

3.3.1 BehaviourUtils Class

The `BehaviourUtils` class provides common methods for managing enemy behaviours in the game. It facilitates actions such as attacking the player, chasing the player, and transitioning to idle states.

Its methods are:

- `updateDungeon(GameManager gameManager, StandardEnemy standardEnemy, Position targetPosition)`: updates the dungeon state to reflect the movement of an enemy to a target position. The method internally utilizes a separate thread to handle game event listener calls asynchronously.
- `playerIsAlive(GameManager gameManager)`: checks if the player is still alive.
- `attackIfPlayerIsNear(StandardEnemyAgent standardEnemyAgent)`: it transitions to `AttackingBehaviour` if the player is adjacent to the enemy. If the enemy is not a boss, it also adds a `RequestReinforcementBehaviour`.
- `chaseIfPlayerIsInRange(StandardEnemyAgent standardEnemyAgent, int range)`: it moves the enemy to chase the player if it is within a specified range.
- `stopIfTargetPositionReached(StandardEnemyAgent standardEnemyAgent, Position targetPos)`: it stops the enemy's movement and transitions to `IdleBehaviour` if the enemy reaches the target position.

3.3.2 MessageHandlerUtils Class

The `MessageHandlerUtils` class provides common methods for handling message triggers, processing inter-agent communication and broadcasting alerts within the game environment.

It is characterized by a `Map` relating trigger strings with message handlers, that are functions that take an `ACLMessage` as input and manipulate it and act accordingly. These handlers are the ones defined in each of the behaviours we have seen previously.

Its methods are as follows:

- `registerHandler(String trigger, Consumer<ACLMessage> handler)`: it registers a handler for a specific message trigger by adding it to the map.
- `handleMessage(ACLMessage message)`: it handles an incoming message by invoking the appropriate registered handler based on the message's trigger type.
- `extractMessageData(ACLMessage message)`: it extracts the data portion of the message, assuming the format to be `TRIGGER:data`.
- `findAgentsByState(StandardEnemyAgent standardEnemyAgent, EnemyState state)`: it finds all the agents with a given state based on their registered state in the DF.

- `broadcastAlert(StandardEnemyAgent sender, String alertType, List<EnemyState> receiverTypes, Position targetPos, int performative)`: it broadcasts an alert message with the specified performative to other agents based on their states. It leverages the `findAgentsByState` auxiliary method to find all the correct receivers.

3.3.3 MovementUtils Class

The `MovementUtils` class provides common methods for handling movement logic of in-game entities. It provides methods for pathfinding, position validation, and pursuing targets.

Its methods are as follows:

- `isPositionValid(Position pos, GameManager gameManager)`: Validates if a given position is within bounds and walkable in the game or not.
- `isWithinGameBounds(Position pos, GameManager gameManager)`: Checks if a position is within the boundary of the game's current room.
- `findNearestValidPosition(Position startingPos, Position targetPos, GameManager gameManager)`: Finds the nearest valid position to the specified target that the entity can move to. Uses a spiral search pattern, starting from the target position and expanding. Used in the `calculateCoverPosition` method of the `CoverRetreatBehaviour` in the case in which the optimal position is not a valid one to move to.
- `pursueTarget(StandardEnemyAgent standardEnemyAgent, Position enemyPos, Position targetPos)`: Moves the given entity towards the target position of one step. It leverages the `computeNextStepAvoidingObstacles` auxiliary method to compute the immediate next step towards the target position. It delegates the game manager to update the GUI to reflect the movement.
- `computeNextStepAvoidingObstacles(Position current, Position target, GameManager gameManager)`, `computeGCost(Position start, Position current)` and `heuristic(Position a, Position b)` are all methods used to compute the next step towards the target position based on A*-like logic. Evaluates the immediate neighbors of the current position and prioritizes valid moves that minimize the cost to the target. Notice that it performs worse than the original algorithm that takes into account the whole path to the target because my implementation has a "local" knowledge and not a "global" one, but it is more efficient computationally speaking for the same reason. The A* algorithm has been adopted in order to make the agents more "obstacles-aware" and not get stuck.

3.4 Rasa

Since the NLP part is mainly the `RasaBridge` and the demonstrative communication between Java/Jade and Rasa, the chatbot is really simple:

- `nlu.yml`: In this file, there are just simple intents to greet the user and to answer help requests if needed. Here we also have the two intents that allow responding to queries referring to the number of alive enemies and notifying the player that a specific agent has spotted them.
- `domain.yml`: Here, simple responses are defined. The only notable things are the defined slots to store the current number of alive enemies and the name of the enemy that just spotted the player.
- `stories.yml`: Here, really primitive stories are defined just to show a normal and consistent flow of conversation.

Regarding the Rasa integration, now that we have a comprehensive understanding of the entire project, we can delve deeper into what this integration enables us to achieve with Rasa:

- Message exchange between Java and Rasa: We can seamlessly send messages from Java to the Rasa bot, allowing for interactive conversations where messages "ping-pong" between the Java client and the Rasa bot. This is exemplified in the feature where the player can send messages to Rasa through the in-game chat (Figure 2).
- Communication between JADE agents and Rasa: Building on the previous point, this also enables JADE agents to communicate with Rasa using the same interface. For example, when an agent spots the player, it sends a message to Rasa, which then processes the response and logs it in the chat. Additionally, Rasa retrieves the agent's name from the message and uses it in its response (Figure 4).
- Dynamic event tracking and knowledge updates: We can send messages to the Rasa tracker events, allowing us to dynamically update its knowledge based on real-time events in the application. In the game, for instance, we notify Rasa of the number of alive enemies, which updates its understanding of the game world. (Figure 3)

References

[1] A* algorithm. `https://en.wikipedia.org/wiki/A*`$_search_algorithm$

[2] Agents-jade. `https://jade.tilab.com/`

[3] Gui-javafx. `https://openjfx.io/`

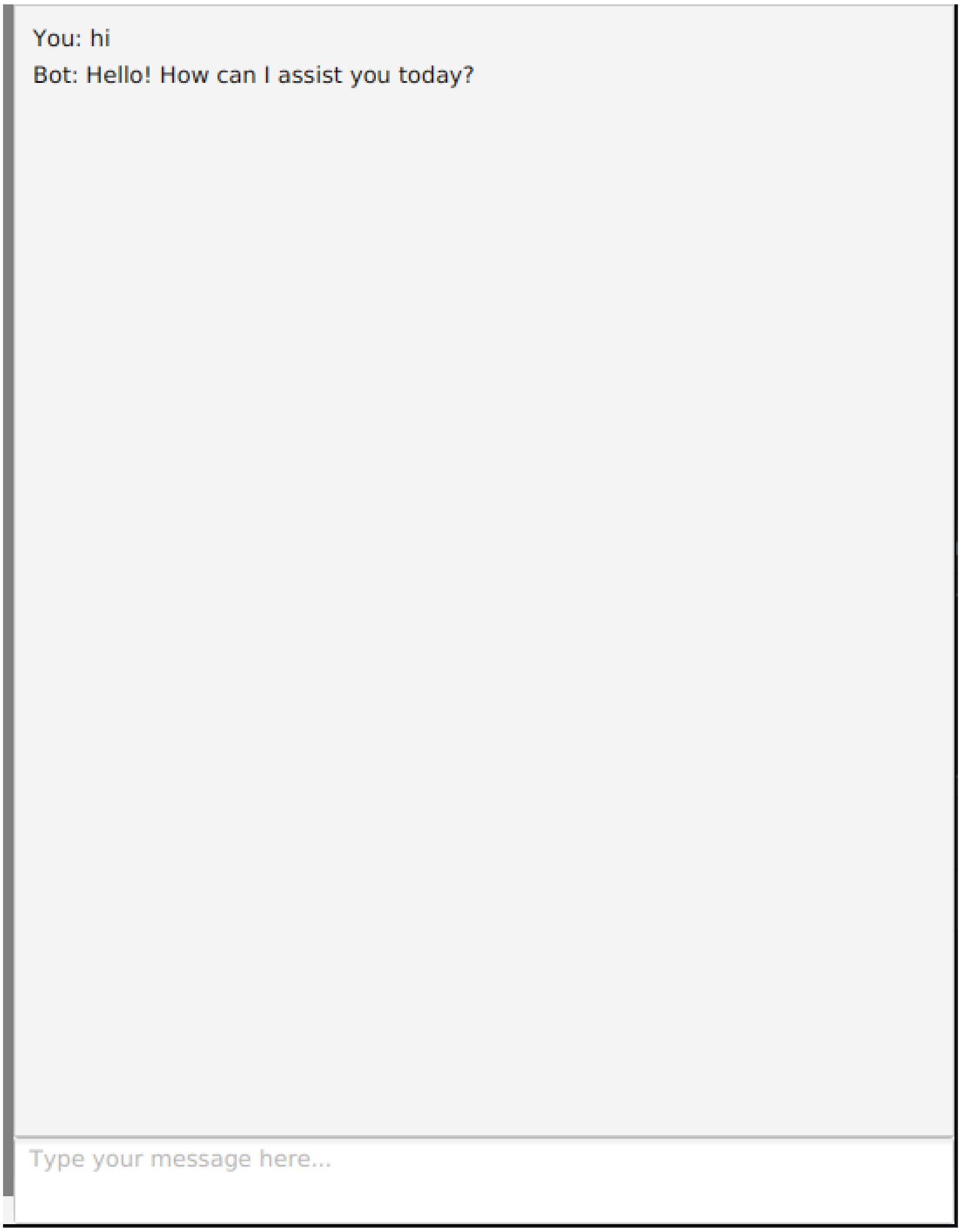
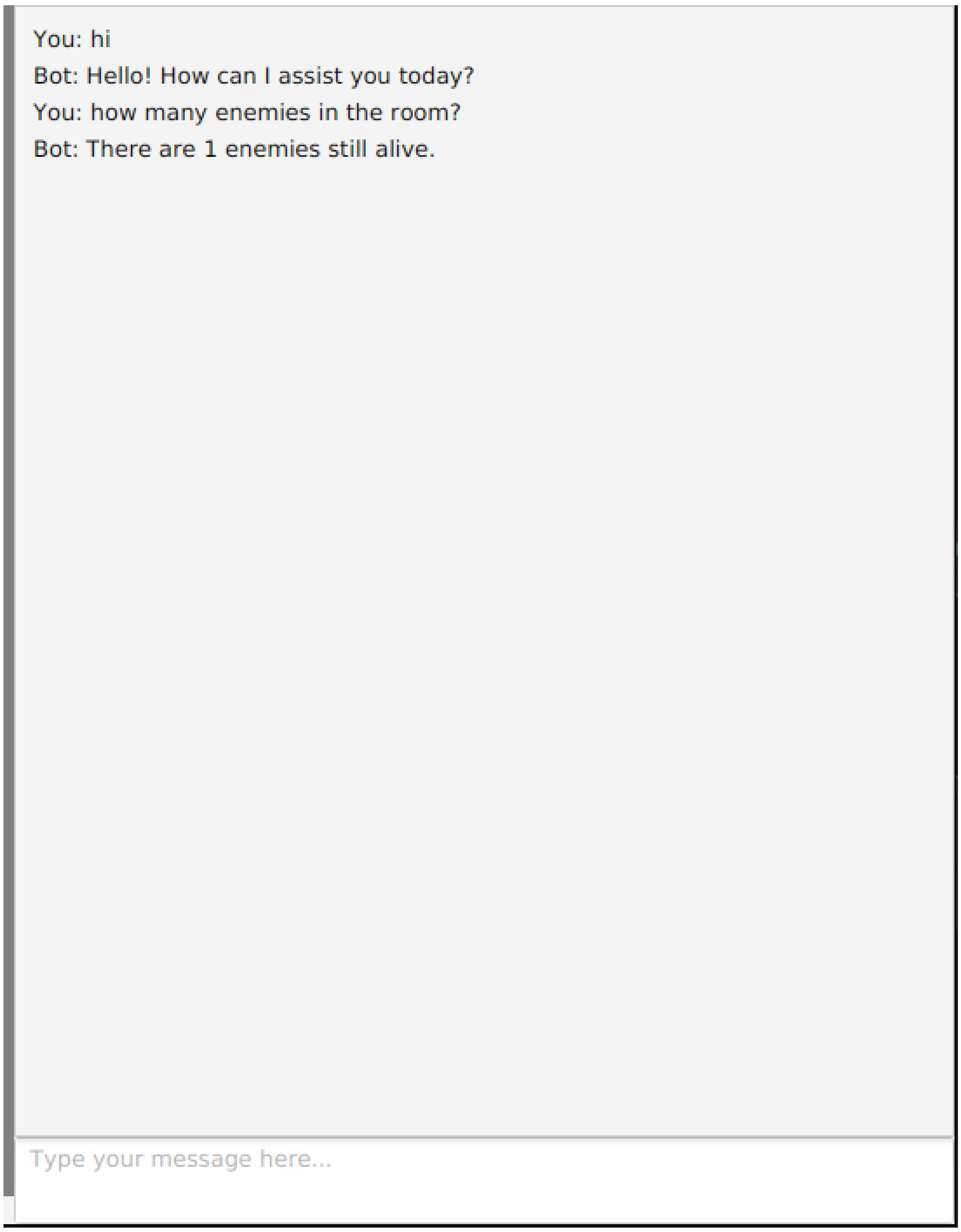
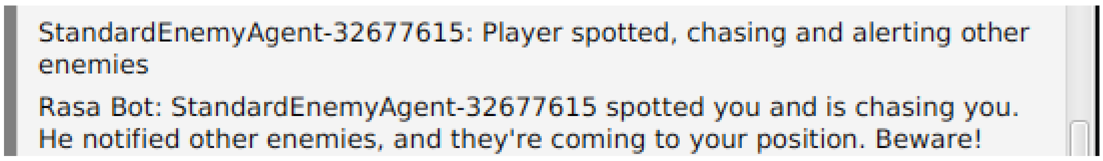

You: hi
Bot: Hello! How can I assist you today?

Figure 2: The Application GUI at start up



```
You: hi
Bot: Hello! How can I assist you today?
You: how many enemies in the room?
Bot: There are 1 enemies still alive.
```

Figure 3: The Application GUI at start up



```
StandardEnemyAgent-32677615: Player spotted, chasing and alerting other
enemies
Rasa Bot: StandardEnemyAgent-32677615 spotted you and is chasing you.
He notified other enemies, and they're coming to your position. Beware!
```

Figure 4: The Application GUI at start up