

MAS Princess - Final Report

Antonio Morelli

`antonio.morelli3@studio.unibo.it`

Fabio Zanotti

`fabio.zanotti2@studio.unibo.it`

February 2025



This project aims to develop a simplified version of the game *Fat Princess*, where two teams of agents with different roles and specific abilities compete to rescue their princess while protecting the one they kidnapped, within a shared environment.

The environment models base layouts, resource locations, and agent states (e.g., health, positions, and tasks) to facilitate strategic interactions.

The overall gameplay is governed by fixed rules with the possibility of dynamic variations based on the game state, adding complexity and adaptability to the system.

Our work provides a platform to explore the dynamics of cooperation and competition in multi-agent systems: by assigning agents distinct roles and objectives, it enables the analysis of emergent behaviors and decision-making processes in controlled scenarios.

This approach also facilitates a deeper understanding of role-based collaboration, studying how agents with complementary capabilities can work together to achieve shared goals.

1 Goal/Requirements

The main goal of this project is to develop a multi-agent system that simulates a simplified version of the game *Fat Princess*, emphasizing strategic interactions, role-based collaboration, and adaptive behavior.

Fat Princess is a multiplayer real-time strategy game where two teams compete to rescue their captured princess from the enemy's base while defending their own. Players assume different roles such as warriors, archers, and priests, working together to fight opponents, gather resources, and influence the outcome through strategic teamwork.

The work is designed to achieve the following objectives:

- **Model a Shared Environment**

- Develop a game world that includes distinct zones, such as the **Base Zone** for defense and princess protection, and the **Battlefield** for resource gathering, combat, and infiltration.
- Represent and track essential components such as resources, agent states (health, position, and roles), and princess attributes.

- **Implement Diverse Agent Roles**

Design agents with specialized roles and behaviors, including:

- **Warriors:** Engage in short-range combat to protect their base and facilitate the rescue of their princess.
- **Archers:** Provide long-range combat support during offensive and defensive operations.
- **Priests:** Heal other agents to maintain team strength during battles.
- **Gatherers:** Collect resources to repair defenses.

- **Explore Cooperation and Competition**

- Investigate how agents interact and collaborate within teams to achieve shared objectives, such as protecting their princess or infiltrating the enemy base.
- Analyze competitive dynamics as opposing teams vie for resources and strategic advantages.

- **Incorporate Adaptive Game Mechanics**

- Implement fixed game rules with the potential for dynamic variations based on the game state, creating opportunities for strategic adaptation and complexity.

1.1 Scenarios

Users can either execute the configuration contained in a `.mas2j` file or dynamically create and execute a new one through interacting with a graphical interface that allows

them to configure the composition of the two opposing teams (Red and Blue). Each team can have up to four members, selected from four distinct agent classes: gatherer, archer, warrior and priest. By customizing team compositions, users can test various strategies and observe how different agent combinations impact the simulation's outcome.

They can then observe the agents' behaviors, interactions, and overall performance within the environment through its visual representation, which helps in understanding emergent behaviors and strategic effectiveness. By iterating over different configurations and adjusting agent compositions based on previous outcomes, users can get deeper insight into the dynamics of agent cooperation, competition, and resource management.

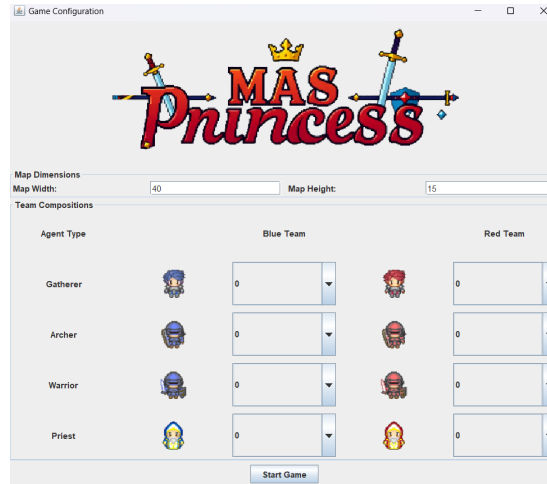


Figure 1: The Configuraton Window.

1.2 Self-assessment policy

The software should run without crashes, and agents should behave according to their intended logic, while the code should be well-structured, following best practices in software development and making it easy to extend and modify. The system should handle agent interactions and environment updates efficiently, avoiding unnecessary computational overhead.

Users should be allowed to configure teams, launch the simulation, and observe the results as intended; all implemented features (agent behavior, environment interactions) should work correctly. The project aims to provide an interactive experience where users can explore different strategies and understand the impact of team composition on simulation outcomes. The ability to test different configurations and observe emergent behaviors determines the system's effectiveness: given the same initial configuration, similar patterns of agent interactions should emerge, ensuring consistency in the simulation outcomes.

2 Requirements Analysis

2.1 Implicit Requirements and Hypotheses

2.1.1 Functional Requirements

The system implicitly requires the following:

- **Consistent Agent Behavior:** Agents must always determine their next action correctly based on their internal logic and priorities.
- **Synchronized Multi-Agent Coordination:** The environment must ensure that all agents have access to an up-to-date and consistent game state to prevent desynchronization.
- **Outcome Consistency and Randomness:** The system must ensure that simulation outcomes align with logical expectations based on team composition, strategy, and probabilistic mechanics.

2.2 Non-functional Requirements

- **Stability and Robustness:** The system must handle edge cases, unexpected behaviors, and high agent interactions without crashing or leading to undefined states.
- **Usability:** Users should intuitively configure teams and observe simulation outcomes without technical barriers.

2.3 Best-Suited Model/Paradigm/Technology

While game development is typically best suited for dedicated game engines like Unity or Godot, which offer optimized tools for 2D rendering, physics, and AI behaviors [2], this project deliberately explores the capabilities of AgentSpeak(L) and Jason together with Java in handling game-like mechanics. Unlike traditional game engines, Jason provides a structured multi-agent system (MAS) framework, where agents reason based on beliefs, desires, and intentions (BDI). This allows for emergent behaviors and decision-making that are more transparent and adaptable than scripted AI at the cost of a more complex and less intuitive implementation.

2.4 Abstraction Gap

Traditional game engines abstract away many low-level concerns, providing built-in support for graphics, physics, and AI scripting [2]. In contrast, AgentSpeak(L) and Jason focus on high-level agent reasoning, requiring additional effort to implement fundamental game mechanics such as movement, interaction, and real-time updates.

The abstraction gap lies in the fact that Jason is suited for modeling decision-making and agent autonomy but lacks native support for real-time simulation features like rendering and collision handling. This work attempts to bridge that gap by integrating

MAS principles with game mechanics, testing whether Jason’s agent-based paradigm can effectively model dynamic environments typically handled by game engines. This exploration contributes to a deeper understanding of how MAS technologies can be adapted beyond traditional domains into interactive, game-like contexts.

3 Design

The environment design follows the **Model-View-Controller (MVC)** [1] pattern, which promotes a clear separation of concerns, making the system modular, maintainable, and extensible. The three main components are:

- **Model:** Manages the core simulation logic, including the state of the environment, agents, structures, and resources. It handles game mechanics such as agent movement, combat, resource management, and victory conditions.
- **View:** Responsible for rendering the graphical representation of the environment. It visualizes the map, agents, and dynamic elements, providing real-time feedback to users about the simulation state.
- **Controller:** Acts as an interface between the agents and the model, processing agent actions and updating the model accordingly. It manages percept generation, synchronizes agent-environment interactions, and handles communication between the agents and the environment.

Following the MVC paradigm ensures that modifications to one component (e.g., enhancing agent logic or updating the graphical interface) can be made independently without affecting the others. It also facilitates debugging and future feature integration, as responsibilities are clearly defined.

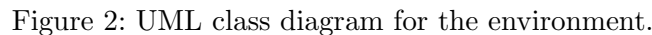
3.1 Project Structure

3.1.1 Environment

The environment consists of a map composed by several elements; base zones serves as the primary area for spawning and as the location of the enemy princess. Bases are symmetrically placed to ensure balanced gameplay, with defensive structures such as walls and gates to control access. Originally, base zones were meant to be smaller and just intended for storing the princess and spawning, but we decided to make them bigger to encourage agent interactions in narrow zones.

The battlefield, located between the two bases, is an open area designed for resource gathering and combat. It features obstacles and structures that create dynamic points of interest and tactical cover during engagements.

Then there are water areas, that act as out-of-map zones and are used to divide the map, introducing constraints that force teams, based on a fixed probability, to choose between two paths: a bridge, from where the agent can slip and die, and a safer land



The logic that governs the functionalities of the map and its elements is entirely contained in the Model class, while the Controller (Environment) class provides the interface that allows interaction between agents and the environment; lastly, the View class handles the visualization of the game state, presenting the agents, objects, and environment to the user in an understandable way.

Defensive elements like walls and gates surround each team’s base. Gates act as controlled entry points to the bases and can be attacked, destroyed, or repaired, with walls forcing agents to use the gate passage. Originally, the idea was to have one gate per team, but we decided to add a second gate to broaden the entry points and reduce the likelihood of agents getting stuck.

Trees are scattered across the battlefield, acting as destructible structures that provide wood when harvested. This resource is essential for repairing gates. Instead of filling the map densely with trees as initially planned, we decided to have trees respawn after

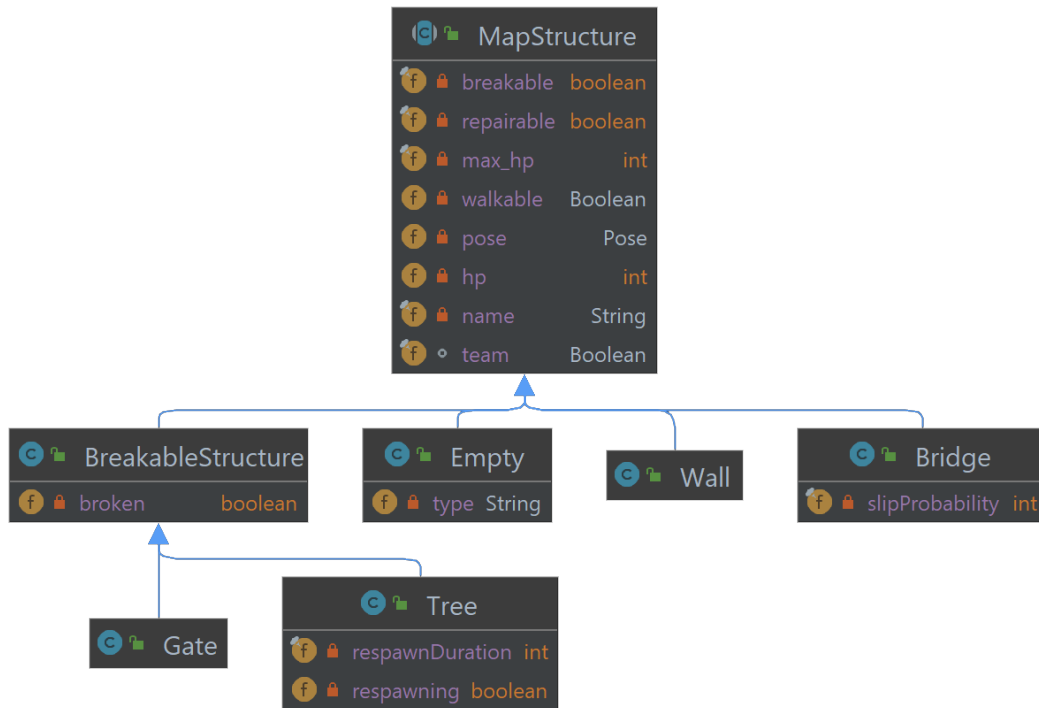


Figure 3: UML class diagram for the structures.

a certain time to leave more space for agent combats and interactions.

Empties are special kind of structures employed for conducting agents movements.

3.2 Resources

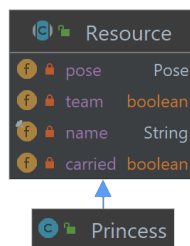


Figure 4: UML class diagram for the resources.

Each team's **Princess** is treated as a resource, an artifact that agents can pick up and carry with them. When carried inside an agent's own base, it determines the victory of that team. We originally considered having more than one resource but ultimately decided to keep only the Princess.

3.3 Agents

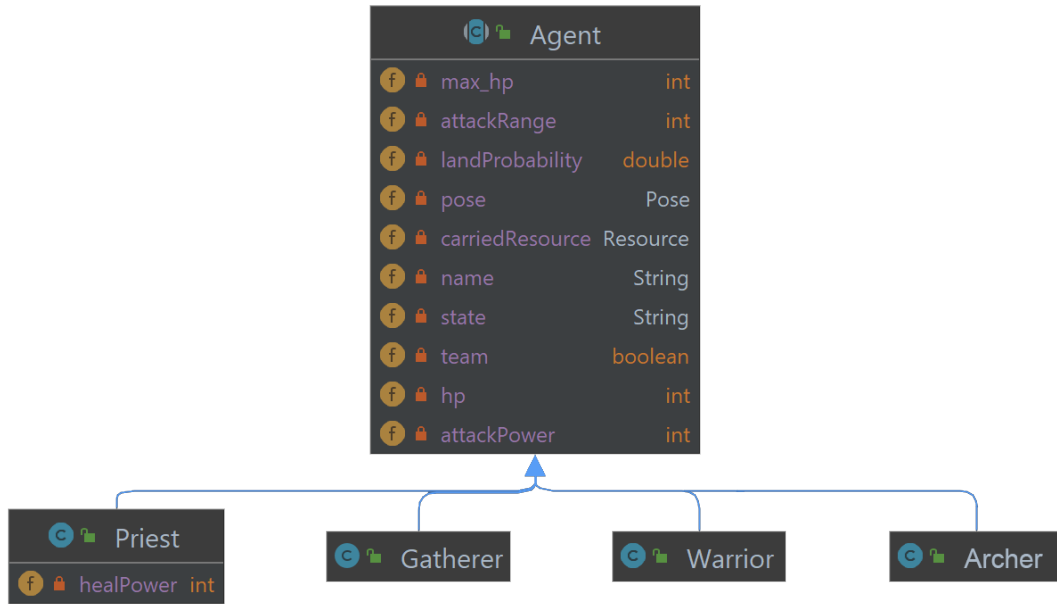


Figure 5: UML class diagram for the agents.

The system features four distinct agent classes, each with specialized roles and behaviors that contribute to team strategies. All agents share common attributes such as health points (HP), attack power, attack range, and position, inherited from a base **Agent** class. However, their unique abilities and characteristics define their roles within the team.

3.3.1 Agent Attributes

Each agent is characterized by the following core attributes:

- **Health Points (HP):** Determines an agent's survivability. Agents with higher HP, like warriors, can withstand more damage.
- **Attack Power and Range:** Defines the damage an agent can inflict and the distance from which they can engage enemies.

- **Critical and missed attack probabilities:** Defines the probabilities that an agent will land a critical attack, multiplying the inflicted damage by five, or miss the attack entirely.
- **Carried Resources** Agents can carry resources. For instance, the Princess, that once picked up have to be escorted back to the team’s base in order to win the game.
- **State and Pose:** Agents maintain an internal state (e.g., spawn, attacking, gathering) and a pose that includes their current position and orientation within the environment.
- **Land Probability:** Represents the likelihood of choosing between the two available crossing of the rivers, the safer land passage or the riskier bridge.

3.3.2 Agent Classes

- **Warrior:** Designed for close-combat situations, warriors have high health and strong attack power, but a limited attack range. Their role is to engage in frontline battles, providing both offensive and defensive capabilities during base defense and princess rescue missions. We didn’t want the warrior to be overpowered by having many HPs and the highest attack value, so we slightly shifted its role towards that of a **tank**—a unit designed to absorb damage and draw enemy attacks, protecting more vulnerable allies while maintaining battlefield presence.
- **Archer:** Archers specialize in long-range attacks with a broader range of perception and slightly higher attack power compared to warriors. However, they have lower health, making them more vulnerable in direct combat. Archers are most effective when positioned behind frontline units, offering support during offensive pushes or defensive holds. They also have the unique ability of attacking enemys beyond walls.
- **Priest:** Priests serve as support units with healing capabilities. They have moderate health and can heal allies within a broader range compared to warriors. While their attack power is low, priests play a crucial role in maintaining team strength during prolonged engagements. We felt the need to add support units like the priest to assist soldiers, promoting more effective and prolonged push towards the enemy base.
- **Gatherer:** Gatherers focus on resource collection and support tasks. They have the lowest health and minimal attack power, reflecting their non-combat-oriented role. Their primary function is to collect resources, such as wood from trees, and then use it to repair gates,



Figure 6: The Warrior, Archer, Priest and Gatherer Sprites for the Blue Team.

3.4 Agents Behaviour

The agent's behaviors are defined using **AgentSpeak(L)** in the respective `.asl` files. They follow a goal-driven approach based on the Belief-Desire-Intention (BDI) model, enabling adaptive and reactive behaviors in response to dynamic environmental changes.

The idea is that each time an agent follows its standard plan, it checks whether percepts from the environment indicate the need for some interaction, such as attacking or healing another agent, repairing a gate, or picking up a princess. If there is nothing to do at the agent's current position, it moves toward the closest objective, determined through environmental reasoning based on the agent's position, state, and the overall game state. Normally, a soldier agent tries to advance toward the enemy base via one of the two paths, preferring the safer one, while the gatherer collects wood to eventually repair doors, acting as a soldier if the wood supply is sufficient. The movement logic is better explained in the implementation details.

Notice that, in this sections, we will include some small references to `.asl` goals and intentions; this is because we drew the boundaries of our project design with trying to encapsulate Jason's theoretical concepts.

3.4.1 Soldier Agent

Given that with the term soldier we refer to both warriors and archers, the central goal of this agent is encapsulated in the recurring goal

!savePrincess. This goal governs all its activities, including offense, defense, and navigation. The agent continuously re-evaluates this goal after each action, creating an adaptive loop that responds to real-time environmental changes. The goal consists of a series of test goals that check for the presence of an entity with which the agent can interact, which is hence set as target; if all the test goals fail, the agent moves towards its current objective, while if at least one succeed, a belief **target(T)** is added to the beliefs base and an action is performed.

Upon initialization or after dying, it executes the **!spawn** plan, which resets its health and verifies whether its internal state has transitioned from **dead** to **spawn**. Once confirmed, the **!savePrincess** goal is reactivated, enabling the soldier to seamlessly re-enter the game after defeat.

The first action performed by any agent is checking whether the game has already concluded, either in victory or defeat for its team. To ensure survival, the agent continuously monitors its health through the **?check_hp** test goal. If its health reaches zero or below, the respawn process is triggered, allowing it to effectively manage its lifecycle within the environment. When an enemy is detected within range, the soldier tests the

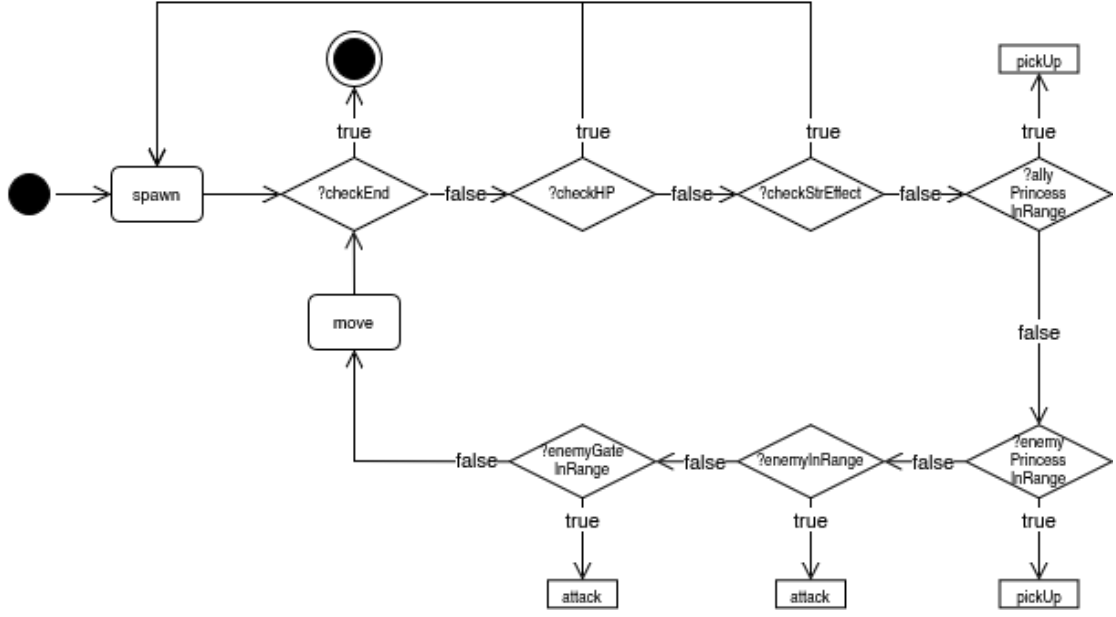


Figure 7: Soldiers (i.e., warriors and archers) standard plan. When a test goal succeeds, the plan loops back.

`?enemyInRange` goal, which dictates its combat behavior. The attack mechanism is facilitated by a **messaging system** in which the soldier, upon selecting a target, sends an `update_hp` message to the enemy agent. This message contains the calculated damage value, which the target processes internally to update its health. The damage calculation incorporates probabilistic elements such as critical hits, which cause increased damage, and misses, which result in no effect. This asynchronous messaging system allows agents to interact without centralized control.



Figure 8: Combat interactions between agents.

Beyond combat, soldiers interact with various elements in the environment. They dynamically identify objects within their perceptual range, such as the allied princess (`?allyPrincessInRange`), the enemy gate (`?enemyGateInRange`), and the enemy princess

(?enemyPrincessInRange). Upon detecting any of these entities, the agent takes appropriate actions according to its strategic objectives.

When no specific interactions are triggered, the soldier executes the !moveTowardsObjective plan, which governs its navigation. Movement is performed following the approach outlined in the implementation details section, ensuring that the agent consistently advances toward its strategic goals.

3.4.2 Priest Agent

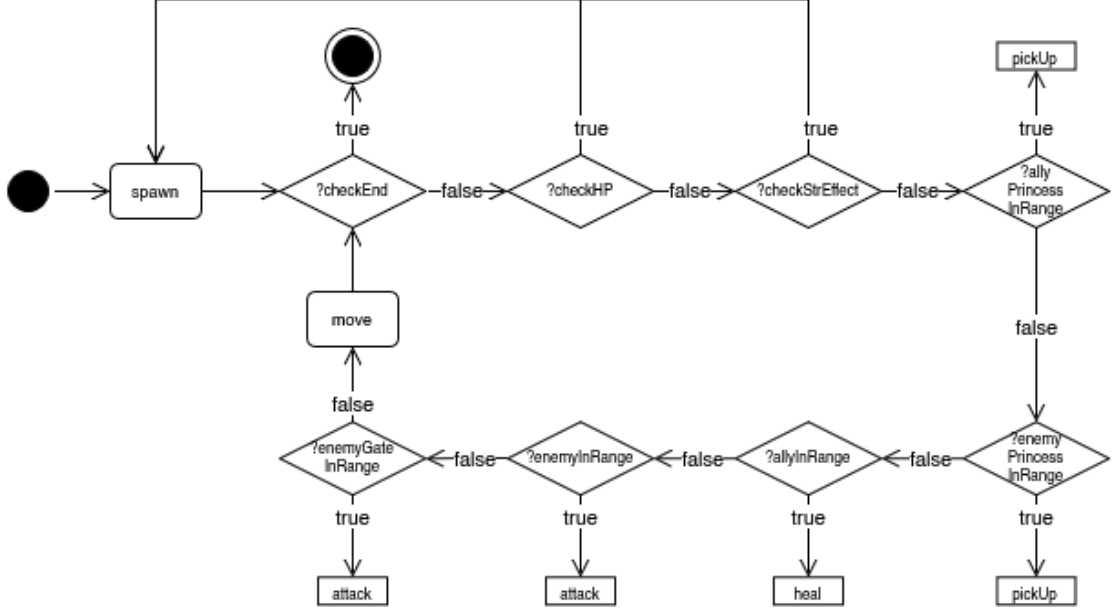


Figure 9: Priest standard plan. When a test goal succeeds, the plan loops back.

The priest agent shares the same behavioral patterns as the soldiers but introduces unique healing mechanics that enhance its strategic role within the team.

The test goal that checks for teammates in need of healing (?allyInRange) is executed before checking for enemies, meaning the agent prioritizes healing actions over offensive tasks. While still capable of engaging enemies, the priest has significantly lower attack power compared to the warrior or archer. Its primary focus remains on support, with combat actions serving as secondary objectives when no healing is needed.

Due to its supportive role, soldiers can engage in longer battles when backed by priests, reducing the need for frequent respawns. Although there is no explicit communication protocol, the priest’s healing behavior indirectly coordinates with frontline units. Warriors naturally draw aggro, and when priests follow closely to provide support, it emerges a “tank-healer” synergy commonly seen in team-based games.

3.4.3 Gatherer Agent

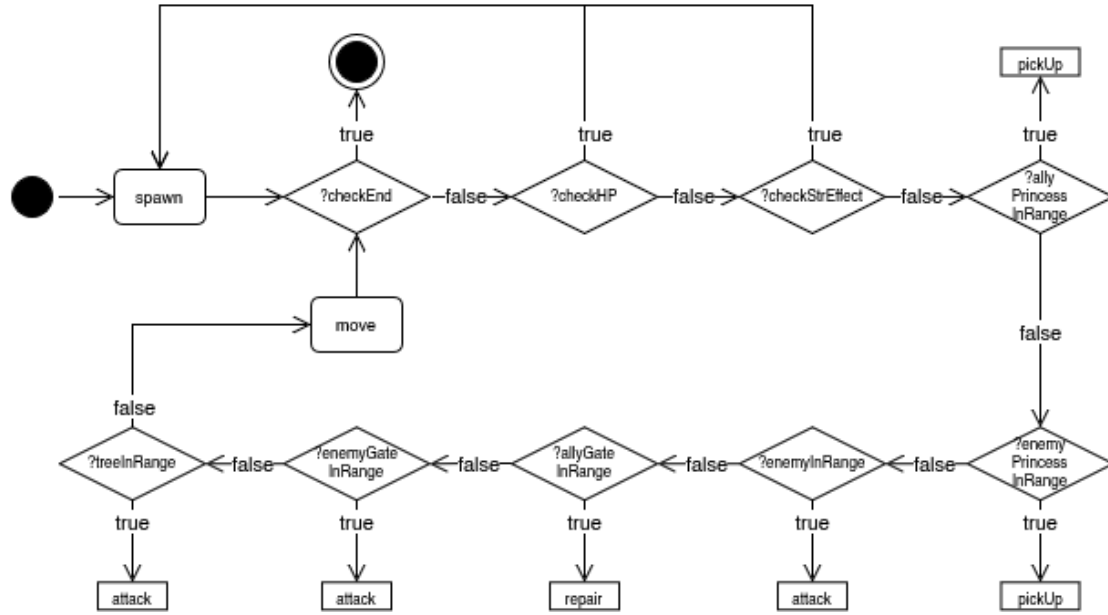


Figure 10: Gatherer standard plan. When a test goal succeeds, the plan loops back.

While the gatherer agent shares foundational behaviors with soldiers—such as movement, enemy engagement, and objective-driven tasks—it introduces distinct functionalities focused on resource management and environmental interaction. These resource-oriented tasks expand the gatherer’s role beyond combat, making it an essential support unit within the team’s strategy.

The gatherer’s primary responsibility is collecting and managing wood, a functionality absent in the warrior’s behavior. Instead of heading toward the enemy base, this agent sets its objective as the closest tree that is not destroyed. Using the `?treeInRange` test goal, it checks for trees within its perceptual range and, if found, triggers the `attack.tree` action to harvest wood, a critical material for repairing gates.

If at least one of the two team gates is destroyed and sufficient resources are available for repairs, the gatherer’s objective shifts to that gate. Upon reaching it, the `?allyGateInRange` test goal succeeds, and the repair action is executed. This defensive capability helps maintain the integrity of the team’s base, reducing its vulnerability to enemy attacks.

It is worth mentioning that if the gatherer has already collected twice the amount of wood required to repair a gate and no gates are currently destroyed, it behaves exactly like a soldier. Although primarily a non-combatant, the gatherer retains basic offensive capabilities to defend itself when threatened. However, due to its low attack power and health, it prioritizes avoidance and support roles over direct confrontation.



Figure 11: Red Team's Gatherer attacking trees to collect wood.

3.4.4 Princess Scenarios

All agents in the system follow the standard plans previously described. However, their behavior deviates whenever a princess—either allied or enemy—is picked up.

An agent that retrieves the allied princess from enemy base will immediately attempt to return to its base, selecting one of the two available paths. To prevent bottlenecks, allied agents will follow the carrier only after its horizontal coordinate surpasses theirs.

Conversely, agents from the opposing team will react immediately if the princess they are holding hostage is stolen. They will pursue the agent carrying her, attempting to recover the princess and prevent a loss.

3.5 Interaction and Environmental Dynamics Beyond the Map

3.5.1 Percept Generation and Management

Percepts are the primary means through which agents acquire information about their environment. The environment generates three categories of percepts:

- **Personal Belief Percepts:** These include an agent's own position, state and current objective. The environment continuously updates these beliefs to reflect changes in the agent's state, allowing for real-time decision-making.
- **Surrounding Percepts:** Agents receive percepts about immediate neighboring tiles, including obstacles, allies, enemies, structures (like gates or bridges), and environmental effects (e.g., slippery bridges with slip probabilities).
- **In-Range Percepts:** These percepts cover objects or agents within an agent's action range. Depending on the agent type, this may include enemy agents, gates, trees (for gatherers), and princesses. This selective percept system ensures that agents only process relevant environmental information based on their roles.

3.5.2 Agent-Environment Interaction

The environment acts as a mediator for all interactions between agents and their surroundings:

- **Movement Validation:** Agents propose movement actions, which the environment validates against obstacles, terrain restrictions, and structural effects. This ensures that agents cannot move through impassable terrain or into blocked tiles.
- **Action Processing:** Actions such as attacking, healing, repairing gates, gathering resources, and picking up the princess are processed centrally. The environment verifies the feasibility of each action based on distance, resource availability, and the current state of the target object or agent.
- **Dynamic Feedback Loop:** After an action is executed, the environment updates both the internal simulation state and the visual representation, providing immediate feedback to agents through updated percepts.



Figure 12: The complete View interface during a simulation.

3.5.3 Agent-Agent Interaction

Although most of the attack and healing logic is executed through the environment, the core of these two actions is performed via messaging. When an agent performs an attack or healing action, it also sends a message to the target agent, informing it of an impending change in its health status. The target agent then autonomously checks whether its health points have reached zero. If so, it triggers the respawn mechanism accordingly.

4 Implementation Details

All relevant implementation details, including class structures, method descriptions, and interactions between system components, are thoroughly documented in the accompanying **Javadoc** generated for the project. The Javadoc provides comprehensive insights into the codebase, covering:

- Detailed class hierarchies and their relationships.
- Method functionalities, input parameters, and return types.
- Descriptions of key algorithms implemented within the system.
- Documentation of utility functions and agent behaviors

Nevertheless, we would like to spend some words on the movements logic.

4.1 Movement logic

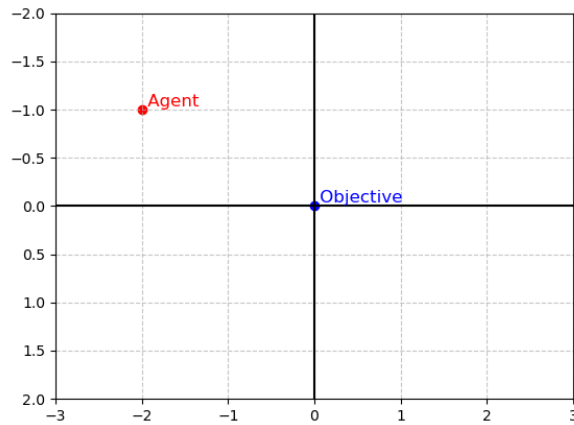


Figure 13: Agent movement.

The movements towards an objective are governed by simple yet efficient hand-crafted rules. A reference system, where the origin is set at the closest objective, determines the direction of the agent's next movement. For instance, in Figure 13, both coordinates of the agent are smaller than those of the objective (inverted y-axis). Therefore, the agent will either move *right* or *down*. The probability of choosing one of these movements is proportional to the distance along the respective axis. In this specific case, the distance along the x-axis is 2, while the distance along the y-axis is 1. Hence, a horizontal movement is more likely to occur.

To compute the movement percentages, let d_x and d_y represent the distances along the x-axis and y-axis, respectively, and let d_{total} be the sum of these distances. The probability for each movement direction is given by:

$$P(horizontal) = \frac{d_x}{d_{total}}, \quad P(vertical) = \frac{d_y}{d_{total}},$$

where $d_{total} = d_x + d_y$.

Using the example provided, where $d_x = 2$ and $d_y = 1$, the probabilities are computed as follows:

$$P(horizontal) = \frac{2}{2+1} = \frac{2}{3}, \quad P(vertical) = \frac{1}{2+1} = \frac{1}{3}.$$

If the cell corresponding to a specific movement is blocked (e.g., by an obstacle), the probability for that direction is set to zero. If only one direction is valid, the agent moves in that direction. However, if both directions are blocked, the agent randomly selects one of the opposite directions (e.g., *left* or *up* in this case) with equal probability. Although the possibility of getting stuck remains, especially in narrow areas such as gates or river passages, the agents are still capable of navigating the environment intelligently.

5 Self-assessment / Validation

5.1 Functional requirements

5.1.1 Consistent Agent Behavior

Agents generally behave according to their intended logic. Core behaviors such as movement, combat, resource gathering, healing, and goal pursuit are consistently executed as specified. However, specific edge cases were identified:

- **Pathfinding Issues Near Gates:** Due to narrow passages introduced by gate structures, agents occasionally block allied units attempting to return with the princess, leading to temporary stalemates. This behavior is an emergent side effect of the environment's spatial constraints rather than a flaw in the agents' individual logic.
- **Graphical Glitches:** Some visual artifacts, such as duplicated sprites, occur sporadically. These are linked to limitations in the chosen graphical framework (JFrame), which is not optimized for real-time game rendering. These issues do not affect the underlying simulation logic but impact visual clarity.

5.1.2 Synchronized Multi-Agent Coordination

Unlike traditional discrete-time simulations, the environment relies on explicit synchronization mechanisms to ensure consistency. Agent actions are executed in a multi-threaded environment, and synchronization is enforced using Java's **synchronized** methods. This guarantees that shared resources, such as the environment model or agent

states, are accessed in a thread-safe manner, preventing race conditions and ensuring data consistency during concurrent actions.

Since there are no global time steps, agents operate asynchronously. However, synchronization mechanisms ensure that conflicting actions (e.g., two agents attacking the same target simultaneously) are resolved correctly, maintaining logical coherence in the simulation.

5.1.3 Outcome Consistency and Randomness

Simulation outcomes are consistent with expectations:

- **Imbalanced Teams:** When one team has more agents, it consistently outperforms the opposing team, confirming that the system correctly reflects strategic advantages derived from numerical superiority. In these scenarios, the presence of gatherers repairing gates plays a critical role in prolonging defensive efforts. This repair mechanism buys valuable time for allied agents to eliminate enemies, as attackers are forced to breach the gates multiple times, reducing the effectiveness of their advances.
- **Balanced Teams:** In scenarios with balanced team compositions, outcomes are more variable. The randomness introduced by probabilistic mechanics (e.g., critical hits, bridge slip probabilities) plays a significant role in determining the victor. Additionally, ideal tactical formations—such as priests following warriors closely—significantly enhance offensive pushes. This coordinated behavior improves the survivability of frontline units, making attacks on enemy bases more effective and sustained. These factors contribute to emergent gameplay dynamics without compromising the overall consistency of the system.

5.2 Non-functional requirements

5.2.1 Stability and Robustness

The software has been tested extensively, and no crashes were observed during normal execution. The system handles agent interactions, environment updates, and resource management without triggering runtime errors, even under complex scenarios involving multiple agents and dynamic environmental changes.

However, occasional temporary deadlocks have been observed. These are caused by the combination of **synchronized** methods and thread wait mechanisms, which can lead to situations where agents are momentarily stuck, unable to proceed with their actions. Despite these occurrences, the system eventually recovers without manual intervention, and the simulation continues as expected. This behavior, while not critical, highlights potential areas for optimization in concurrency handling.

5.2.2 Usability

The graphical interface eliminates the need for users to interact directly with the underlying code, providing a smooth execution experience.

5.3 Validation Approach

While formal testing frameworks were not employed extensively, validation was conducted through:

- **Exploratory Testing:** Multiple configurations and edge cases were manually tested to observe agent behavior, environmental interactions, and simulation stability.
- **Outcome Consistency Checks:** Repeated runs with identical initial conditions yielded similar emergent patterns, confirming the system's deterministic behavior in controlled settings, with variability introduced only by intended random mechanics.

6 Deployment Instructions

To install and launch the software follow these steps:

Prerequisites

- **Java 19:** Ensure Java 19 is installed. Verify the installation with:

```
java -version
```

If Java 19 is not installed, download it from <https://www.oracle.com/java/technologies/javase/jdk19-archive-downloads.html> or install it via a package manager.

- **Gradle:** Install Gradle or use the provided Gradle Wrapper. Check if Gradle is installed with:

```
gradle -v
```

If not installed, follow instructions at <https://gradle.org/install/>.

Project Setup

1. Clone or download the project repository.
2. Navigate to the project directory containing the `build.gradle.kts` file:

```
cd MAS_Princess/
```

Build and Run

To execute the pre-launch configuration window, run the following command:

```
gradle runConfigWindow
```

This command compiles the project, resolves dependencies (including the Jason library), and starts the configuration interface.

If you want to run the last produced `.mas2j` file and avoid launching the configuration window, run the following command:

```
gradle runMas_PrincessMas
```

Usage Instructions

Troubleshooting

- If permission issues occur, try:

```
chmod +x gradlew  
./gradlew runConfigWindow
```

- Ensure the `JAVA_HOME` environment variable points to the correct Java 19 installation if Gradle reports Java version issues.

7 Usage Examples

Once the program starts:

- A window will open where the user can set the team compositions (selecting agents for both Red and Blue teams) and adjust the map dimensions.
- After configuring the desired settings, click on the **Start Game** button to begin the simulation.
- When the simulation ends, a pop-up window will appear announcing the winning team.
- The user will then have the option to either start another run or close the program.

8 Conclusions

This project aimed to develop a multi-agent system simulating a simplified version of *Fat Princess*, focusing on strategic interactions, role-based collaboration, and emergent behaviors. We designed and implemented a grid-based environment with dynamic elements such as gates, bridges, and resource structures. Four distinct agent classes—warriors, archers, priests, and gatherers—were created, each with specialized roles and behaviors defined through the BDI model using AgentSpeak(L).

The system integrates real-time perception, probabilistic decision-making, and asynchronous multi-agent coordination, with an emphasis on exploring the capabilities of the Jason framework in handling game-like mechanics. We provided a pre-launch configuration interface to customize team compositions and map settings, enhancing the flexibility and replayability of the simulation.

8.1 Future Works

While the project met its core objectives, several areas could be expanded to improve functionality and depth:

- **Enhanced Pathfinding:** Implementing advanced pathfinding algorithms (e.g., A*) to reduce navigation inefficiencies, especially around chokepoints like gates.
- **Improved Concurrency Management:** Addressing occasional deadlocks caused by synchronized methods and thread handling to ensure smoother agent coordination.
- **Graphical Enhancements:** Replacing the current JFrame-based visualization with a more optimized graphics framework to eliminate visual glitches and improve performance.
- **Richer Agent Communication:** Introducing explicit communication protocols between agents to enable coordinated strategies beyond emergent behaviors.
- **Extended Game Mechanics:** Adding new agent roles, dynamic objectives, and environmental events to diversify gameplay scenarios.
- **Reinforcement Learning Integration:** Exploring the application of reinforcement learning techniques to enable agents to learn and adapt their strategies based on environmental feedback, improving decision-making over time beyond predefined BDI logic.

8.2 What Did We Learn

Throughout the development of this project, we gained valuable insights into multi-agent system design and the practical application of the BDI model. Key takeaways include:

- **Multi-Agent Coordination:** Understanding the complexities of asynchronous agent interactions and how emergent behaviors arise from simple, goal-driven agents.
- **BDI Programming with Jason:** Gaining hands-on experience with AgentSpeak(L) for behavior modeling, highlighting both its strengths in reasoning and limitations in handling real-time game dynamics.
- **Concurrency Challenges:** Learning about thread synchronization in multi-agent environments and the impact of concurrency issues on system stability.
- **Game Design Considerations:** Recognizing how environmental design, resource distribution, and probabilistic mechanics influence strategic gameplay and simulation outcomes.

References

- [1] James Bucanek. “Model-view-controller pattern”. In: *Learn Objective-C for Java Developers* (2009), pp. 353–402.
- [2] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.