Agent Pac-Man

Leo Marzoli
leo.marzoli@studio.unibo.it

August 2024

Project related to Intelligent System Course
at Engeneering and Computer Science University of Bologna, campus of Cesena.

Abstract

In this project, I aimed to develop a game like the famous Pacman game, implemented using Java and a Swing GUI. The project was developed as part of the Intelligent Systems course at the University and aimed to demonstrate the ability to design and develop complex software that can integrate an intelligent agent behaviour, with a particular focus on technologies like JASON.

1 Goals

The main goal of the project was to create a playable version of Pacman that stays true to the original as much as possible. So what differ the most is the developed code. In fact, in this version of Pacman the main goal is to have an agent Pacman that plays itself the game and the four classic agent enemies that try to catch pacman in order to make the player lose. Some of the things that had been chosen to achive in the system are: object-oriented programming principles, implement intelligent agents using AgentSpeak(L) and JASON, develop an intuitive and responsive graphical user interface via Swing.

2 Requirements

The functional and non-functional requirements of the project included:

2.1 Functional Requirements

- The game must allow the player to see the behaviour of Pacman in the game.
- The ghosts must have distinct behaviors based on intelligent agent logics.
- A real-time scoring system must be implemented.
- The game must end when Pacman gets catched or when all points are collected.
- Pacman must be able to get a powerups and eat enemies to get points and simplify the end of the game.

2.2 Non-Functional Requirements

- The game must run on Windows operating systems.
- Input latency must be minimal to ensure a good gaming experience.
- The code must be well-documented and structured to facilitate maintenance.
- Agent technologies must be integrated in a way that they are easy to understand for others.

3 Technologies

The technologies used in the project include:

- **Java**: The main programming language used for game development.
- **Swing**: Library used to create the graphical user interface.
- **AgentSpeak(L) and JASON**: Tools used to create intelligent agents for the ghosts.
- **JUnit**: Framework for unit testing the code.
- **Gradle**: Build automation tool used to manage dependencies and automate the build process.

4 Design

For the development of the project from a Design point of view, a strategy based on MVC patterns was adopted, where respectively there is a "PacmanModel", a "GameView" and a "PacmanLogic" which acts as a controller in this case.

There are different ways of implementing MVC, but in this case it was done through the support of an Observer pattern that notifies the model changes made by the controller and adapts the view accordingly.

The various classes and their role within the project are briefly described below:

4.1 PacmanModel

The `PacmanModel` class represents the main game model for Pacman, responsible for managing the game state, including walls, score points, power-ups, the position of Pacman, and the enemies. It implements the Observer pattern to notify observers about changes in the game state.

4.1.1 Attributes

- `PROBABILITY_TO_SPAWN_SCOREPOINTS` (constant): Probability of spawning score points.
- `NUMBER_OF_POWERUPS` (constant): Number of power-ups in the game.
- `GRID_SIZE` (constant): Size of the game grid.
- `CELL_SIZE` (constant): Size of each cell in the grid.
- `ENEMY_SLIPPERY_PROBABILITY` (constant): Probability that an enemy can escape an action.
- `DURATION_POWERUP` (constant): Duration of a power-up.
- `DURATION_DISABLE_ENEMIES` (constant): Duration of disabling enemies.
- `RAND` (constant): Random number generator.
- `NUM_ENEMIES` (constant): Number of enemies in the game.
- `powerUpEndTime` (variable): End time of the power-up.
- `walls` (variable): Boolean matrix representing the walls.
- `scorePoints` (variable): Boolean matrix representing the score points.
- `powerUps` (variable): Boolean matrix representing the power-ups.
- `pacmanSphere` (variable): Position of Pacman.
- `enemies` (variable): Positions of the enemies.
- `score` (variable): Player's score.
- `observers` (variable): List of registered observers.
- `enemyDisableTimes` (variable): Map of enemy disable times.
- `logger` (variable): Logger for recording information.

4.1.2 Methods

- `PacmanModel(int[][] gridMatrix)`: Constructor that initializes the game model based on the provided grid matrix.
- `initializeWalls(int[][] gridMatrix)`: Initializes the game walls based on the grid matrix.
- `initializeYellowSphere()`: Initializes Pacman's position in a random non-wall cell.
- `initializeEnemies(int numEnemies)`: Initializes the positions of the enemies in random non-wall cells.
- `generateScorePoints(double probability)`: Generates score points with a given probability in non-wall cells.

- `generatePowerUps(int maxPowerUps)`: Generates a maximum number of power-ups in non-wall and non-occupied cells.
- `getGridSize()`: Returns the grid size.
- `getCellSize()`: Returns the cell size.
- `getWalls()`: Returns the walls matrix.
- `getScorePoints()`: Returns the score points matrix.
- `getPowerUps()`: Returns the power-ups matrix.
- `getPacmanSphere()`: Returns Pacman's position.
- `getEnemies()`: Returns the enemies' positions.
- `getPowerUpEndTime()`: Returns the power-up end time.
- `getScore()`: Returns the player's score.
- `addScore(int points)`: Adds points to the player's score.
- `incrementScore()`: Increments the player's score by one and notifies the observers.
- `consumeScorePoint(int x, int y)`: Consumes a score point if present at the specified position.
- `consumePowerUp(int x, int y)`: Consumes a power-up if present at the specified position.
- `isScorePoint(int x, int y)`: Checks if there is a score point at the specified position.
- `isPowerUp(int x, int y)`: Checks if there is a power-up at the specified position.
- `isEnemyDisabled(int enemyId)`: Checks if an enemy is disabled.
- `disableEnemy(int enemyId, long duration)`: Disables an enemy for a specified duration.
- `enableEnemy(int enemyId)`: Enables an enemy.
- `isEnemySlippery()`: Checks if an enemy can escape an action with a certain probability.
- `addObserver(Observer observer)`: Adds an observer to the list.
- `removeObserver(Observer observer)`: Removes an observer from the list.
- `notifyObservers()`: Notifies all registered observers.
- `setPacmanSphere(int x, int y)`: Sets Pacman's position and notifies the observers.
- `setEnemyPosition(int index, int x, int y)`: Sets the position of a specified enemy and notifies the observers.
- `setPowerUpEndTime(long powerUpEndTime)`: Sets the power-up end time.
- `areAllScorePointsCollected()`: Checks if all score points have been collected.

4.1.3 Role in the System

The `PacmanModel` class plays a crucial role in the system as it maintains the internal state of the game, manages the logic related to collecting points and power-ups, and coordinates interactions between Pacman and the enemies. It implements the Observer pattern to ensure that changes in the game state are propagated to all views or other components that need to be updated. Additionally, it provides methods to initialize and update game elements, enabling the implementation of complex game dynamics.

4.2 PacmanLogic

The `PacmanLogic` class encapsulates the game logic for Pacman, managing Pacman's movements, interactions with the game environment, and the behavior of enemies. It works closely with the `PacmanModel` to update and maintain the game state.

4.2.1 Attributes

- `pacmanModel` (variable): The game model containing the current state of the game.
- `logger` (variable): Logger for recording information and debugging.
- `gameRunning` (variable): A flag indicating whether the game is currently running.
- `powerUpScheduler` (variable): A scheduler for handling power-up timing.

4.2.2 Methods

- `PacmanLogic(PacmanModel pacmanModel)`: Constructor that initializes the game logic with a given game model.
- `boolean move(Direction direction)`: Moves Pacman in the specified direction if it is a valid move. Consumes score points and power-ups, checks victory conditions, and handles interactions with enemies.
- `void handlePowerUp()`: Activates the power-up effect for a limited time and schedules its deactivation.
- `void disableEnemy(int enemyId)`: Disables an enemy for a specified duration and schedules its reactivation.
- `boolean isValidMove(int newX, int newY)`: Checks if moving to the specified coordinates is valid (within bounds and not a wall).
- `Direction choosePreferredDirection(Point currentPos, int distance)`: Chooses the preferred direction for Pacman based on the presence of score points or power-ups.
- `Point getNewPosition(Point currentPos, Direction direction)`: Calculates the new position based on the current position and direction.
- `void moveEnemy(int enemyId, Direction direction)`: Moves an enemy in the specified direction if it is a valid move and the enemy is not disabled or blocked.
- `Direction chooseDirectionTowardsPacman(Point enemyPos, Point pacmanPos)`: Chooses the direction for an enemy to move towards Pacman's position.
- `void checkPacmanCapture()`: Checks if Pacman has been captured by any enemies. If so, it handles the capture or disables the enemy if Pacman has an active power-up.

4.2.3 Role in the System

The `PacmanLogic` class is essential for the functioning of the game as it implements the core logic for Pacman's movements, interactions, and the behavior of enemies. It ensures that all actions taken by Pacman and enemies are valid and updates the game state accordingly. This class also manages the activation and deactivation of power-ups and handles the capture logic, which is crucial for maintaining the gameplay dynamics. By interfacing with the `PacmanModel`, it ensures that the game state is consistently updated and that all changes are accurately reflected in the model.

4.3 Pacman_Agent

The `pacman_agent.asl` file defines the behavior of the Pacman agent in the Pacman game. It sets the initial goal for the agent and outlines the plans for moving Pacman either preferentially or in a specified direction. The agent continuously attempts to move preferentially, ensuring that Pacman actively seeks out score points or power-ups based on the game logic. By sending move actions to the environment, the agent interacts with the `Arena2DEnvironment` class, which processes these actions and updates the game state accordingly. This file is crucial for the autonomous behavior of Pacman, enabling it to navigate the game environment effectively. From the BDI perspective here's the description of the agent flow:

- **Initial Goal**: The `!start` goal is immediately activated when the agent starts. Pacman announces its presence and then moves according to a preferential strategy using the `!move(preferential)` goal.
- **Plans**:
  - `+!start` Plan: The agent begins with a greeting and sets its next goal to move in a preferential direction.

- `+!move(preferential)` Plan: Pacman attempts to move preferentially, likely using some predefined strategy for selecting an optimal direction. After attempting a move, the agent waits for 500 milliseconds before setting the goal to move again. This creates a loop where Pacman continuously attempts to move preferentially.
- `+!move(Direction)` Plan: If a specific direction (not preferential) is given, Pacman will move in that direction. After completing the action, Pacman waits for 500 milliseconds before returning to its preferential movement plan.

- **Behavior Summary**: Pacman moves in a loop, predominantly using a preferential movement strategy. It can also be instructed to move in a specific direction via the `+!move(Direction)` plan. The `.wait(500)` ensures a pause between actions, simulating real-time movement behavior.

4.4 Enemy_Agent

The `enemy_agent.asl` file defines the behavior of the enemy agents in the Pacman game. It sets the initial goal for the agents and outlines the plans for initializing and updating the positions of the enemies. The enemies are initialized sequentially, and each enemy's position is updated continuously, with movements in random directions. By sending move actions to the environment, the agents interact with the `Arena2DEnvironment` class, which processes these actions and updates the game state accordingly. This file is crucial for the dynamic behavior of the enemies, enabling them to actively pursue Pacman within the game environment. From the BDI perspective here's the description of the agent flow:

- **Initial Goal**: The `!start` goal is activated when the enemies are initialized. The enemies announce their presence and then initialize all enemy positions using the `!init_all_enemies` goal.

- **Plans**:
  - `+!start` Plan: Each enemy announces that it has started and proceeds to initialize all enemies.
  - `+!init_all_enemies` Plan: This plan initializes the positions of four enemies by iterating through enemy IDs (`0`, `1`, `2`, `3`). Each enemy is then updated individually through the `!init_enemies(Id)` goal.
  - `+!init_enemies(Id)` Plan: For each enemy, this plan updates the enemy's initial position (which can be modified later), waits for 500 milliseconds, moves the enemy randomly, and then recursively calls itself to continuously reinitialize the enemy. This creates a loop of random movement for each enemy.

- **Behavior Summary**: The enemies continuously move in random directions, but each enemy can be individually initialized and updated. Their movement is also controlled through recursive planning with `.wait(500)` to simulate real-time behavior.

4.5 Design Summary (BDI Concepts)

- **Beliefs**: Both Pacman and enemies hold beliefs about their positions and possible directions of movement. For example, `+enemy_position(Id, 0, 0)` represents the enemy's belief about its current location.

- **Desires**: Pacman's desire is to keep moving, especially in a preferential manner. Enemies' desire is to keep moving randomly, initialized individually.

- **Intentions**: Both agents follow through on their intentions, either moving preferentially for Pacman or moving randomly for enemies, and they continuously re-execute their movement plans.

4.6 Arena2DEnvironment

The `Arena2DEnvironment` class extends the `Environment` class and serves as the environment for the Pacman game, managing the initialization, actions, and termination of the game. It interfaces with the `PacmanLogic`, `PacmanMap`, and `PacmanModel` classes to maintain the game state and respond to agent actions.

4.6.1 Attributes

- `DISTANCE_TO_CHASE_PACMAN` (constant): The distance within which an enemy will chase Pacman, set to 5 cells.

- `PACMAN_DIST_TO_CHECK_ENV` (constant): The distance within which Pacman will check the environment for score points or power-ups, set to 2 cells.

- `logger` (variable): Logger for recording information and debugging.
- `pacmanLogic` (variable): Instance of `PacmanLogic` to manage game logic.
- `pacmanMap` (variable): Instance of `PacmanMap` to manage the game map.
- `pacmanModel` (variable): Instance of `PacmanModel` to maintain the game state.

4.6.2 Methods

- `void init(final String[] args)`: Initializes the environment by setting up the map, model, and logic, and starts the game.
- `boolean executeAction(String agName, Structure action)`: Executes actions performed by agents, such as moving Pacman or an enemy. It checks the validity of moves and updates the game state accordingly.
- `void stop()`: Stops the environment and terminates the game.

4.6.3 Role in the System

The `Arena2DEnvironment` class is essential for the functioning of the Pacman game as it sets up the game environment and manages interactions between the game logic and the agents. It initializes the game components, processes agent actions, and ensures the game state is updated accordingly. This class also handles the termination of the game, ensuring all resources are properly managed. By interfacing with the `PacmanLogic`, `PacmanMap`, and `PacmanModel`, it maintains a cohesive and interactive game environment.

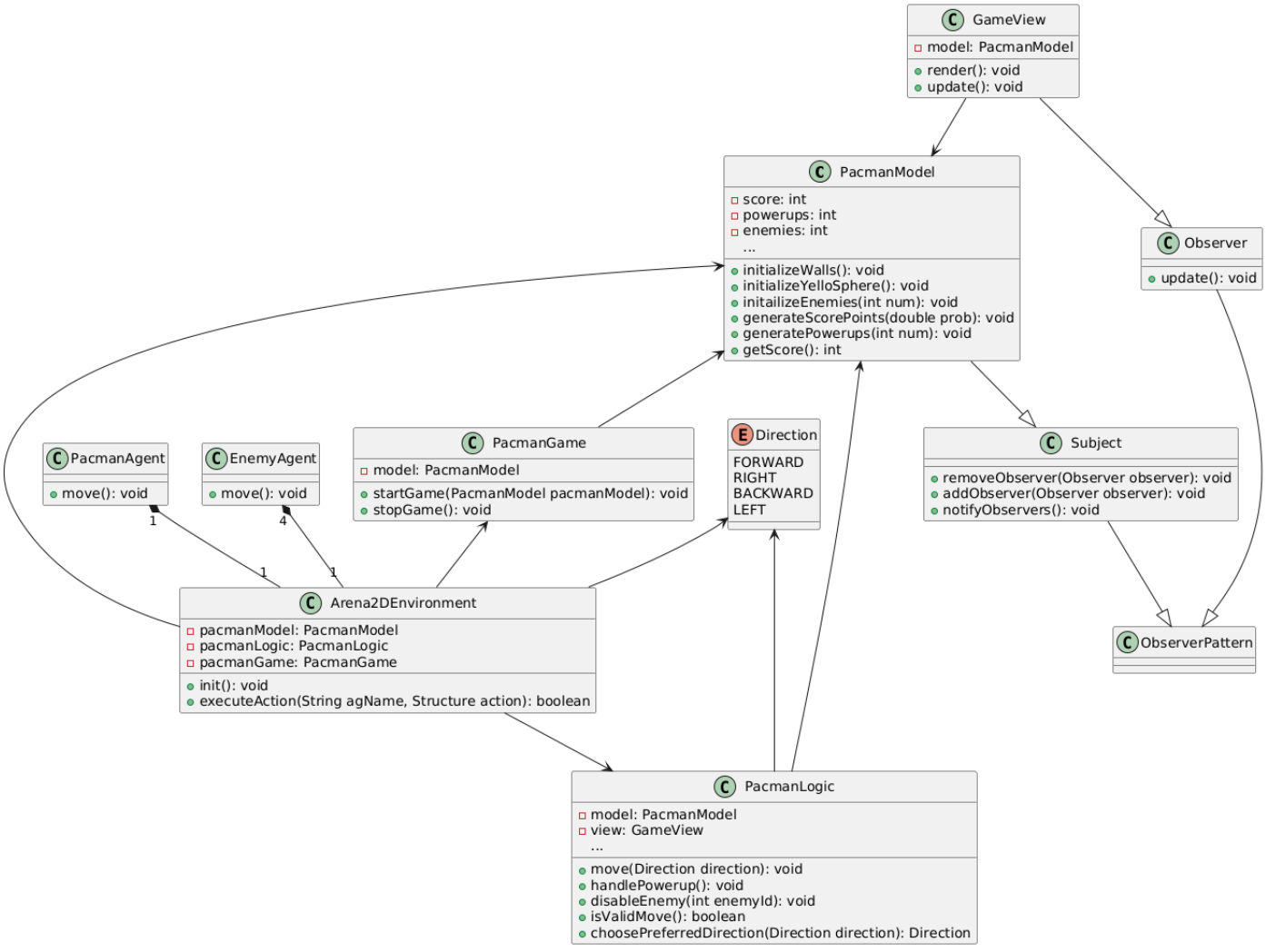

Fig. 1 - Class Diagram of the above mentioned classes.

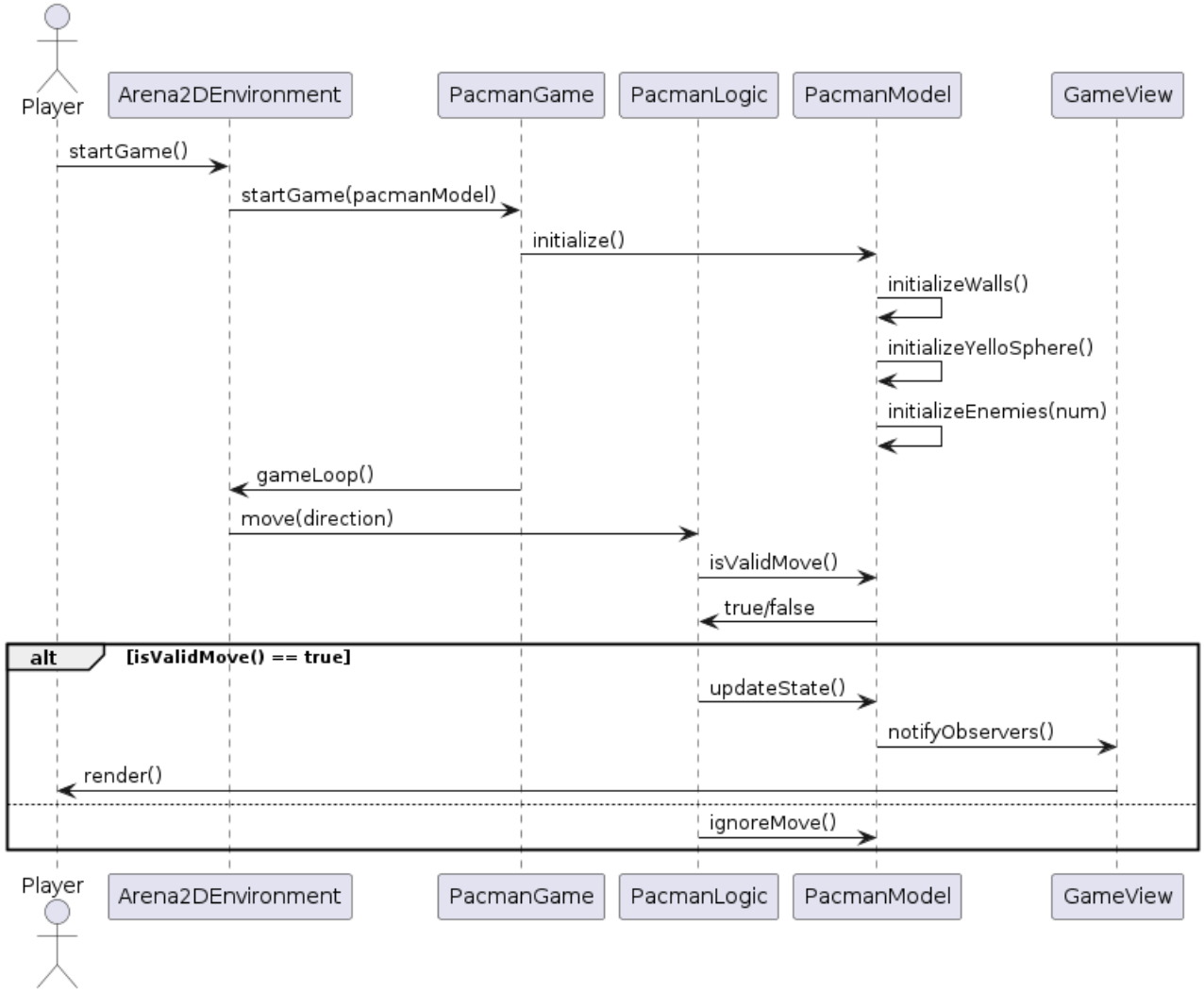

Fig. 2 - Sequence Diagram that shows the basic interactions among the classes.

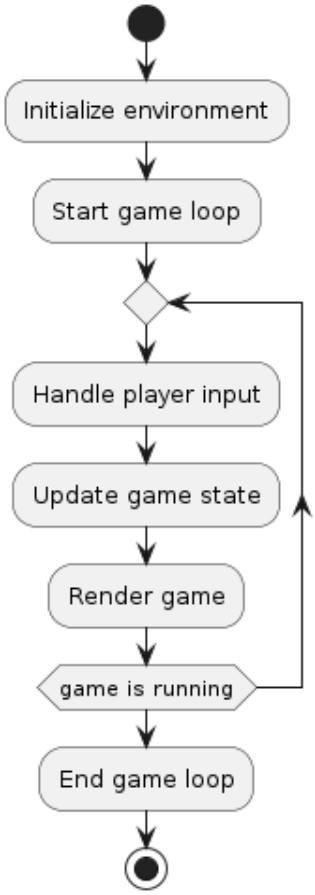

Fig. 3 - Activity Diagram of the Arena2DEnvironment class.

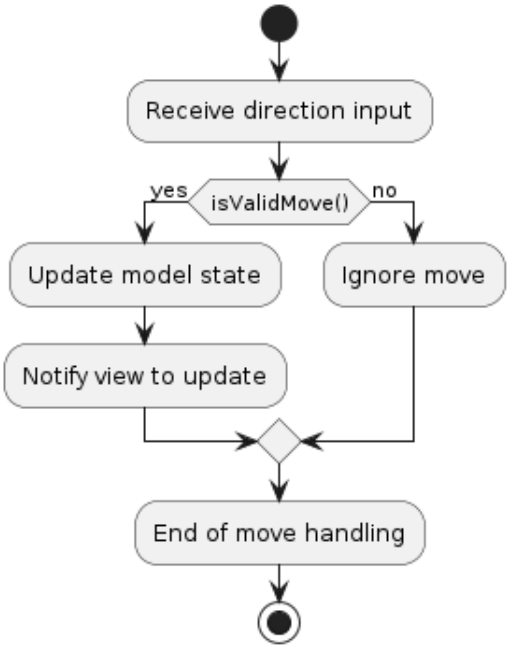

Fig. 4 - Activity Diagram of the PacmanLogic class.

5 Self Assessment

The project was a success from several perspectives:

- **Code Quality**: The code is well-structured and commented, facilitating future maintenance and extension.
- **Implemented Features**: All planned features were successfully implemented.
- **Learning**: The project allowed for a deeper understanding of Java programming, graphical interface development, and intelligent agent algorithms.
- **Testing**: The project has been tested and checked for his working during its develop.

About testing here i provide a screen of the test implemented:

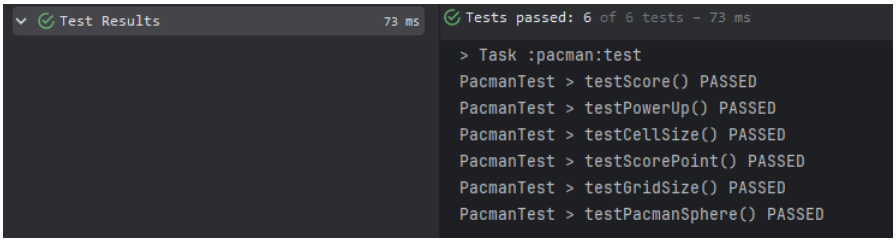

Fig. 6 - Test of the PacmanLogic in order to check if main features works.

To run the tests yourself if needed , you need to download the project folder then open it in your prefered IDE (i suggest Intellij IDEA) and in the editor type the following command:

```
./gradlew test
```

6 Deployment

To deploy the project I have used a build tool Gradle that simplify the distribution and the download of dependecies that had been developed. You just need to follow these steps:

- Download and Open the GitHub project in any IDE (Intellij IDEA prefered).
- From the command line, execute `./gradlew runPacmanMas` to run the game.

7 Usage

As said in the deployment to start the game, open the project in an IDE and execute the following command from the command line:

```
./gradlew runPacmanMas
```

Now since this is an intelligent system's course project the user hasn't a realgameplay due to the fact that it wasn't one of the purpose. The use can watch the pacman play itself, if the pacman gets caught the game window closes with a lose, otherwise if the pacman keeps surviving without getting caught the game closes with a victory only if it has picked all the scorepoints up.

Below there are some figures that shows how the game will look like and the various scenarios, remember that the yellowsphere is pacman and the enemies for "grapich" reason have all the same color in this version of pacman, plus the grey dots are scorepoints and the green dots are powerups, anything else that is blue is a wall so it can't be crossed by pacman neither the enemis and the rest is the walkable zone.

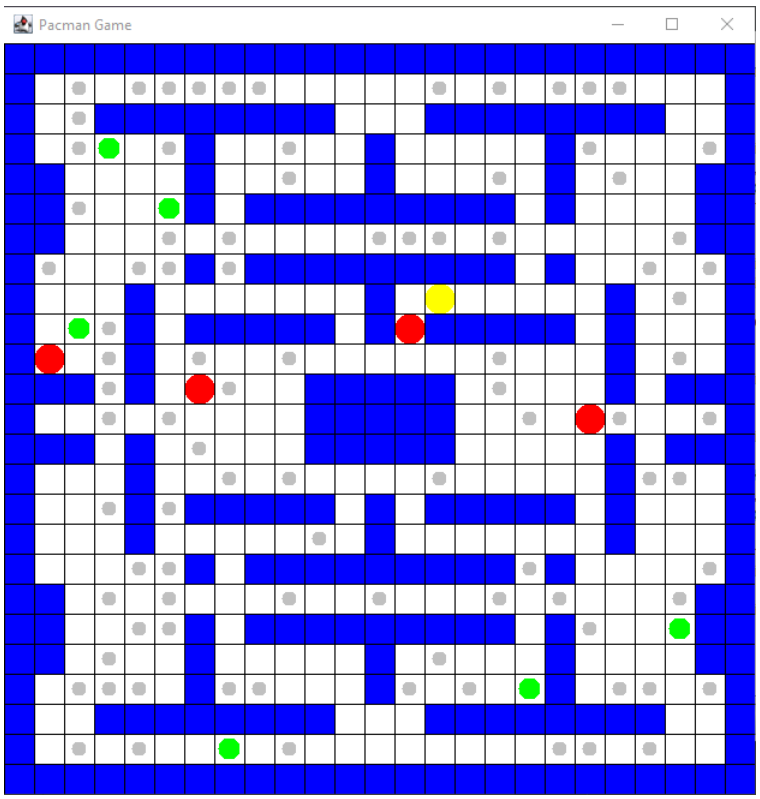

Fig. 7 - GUI with Pacman example 1.

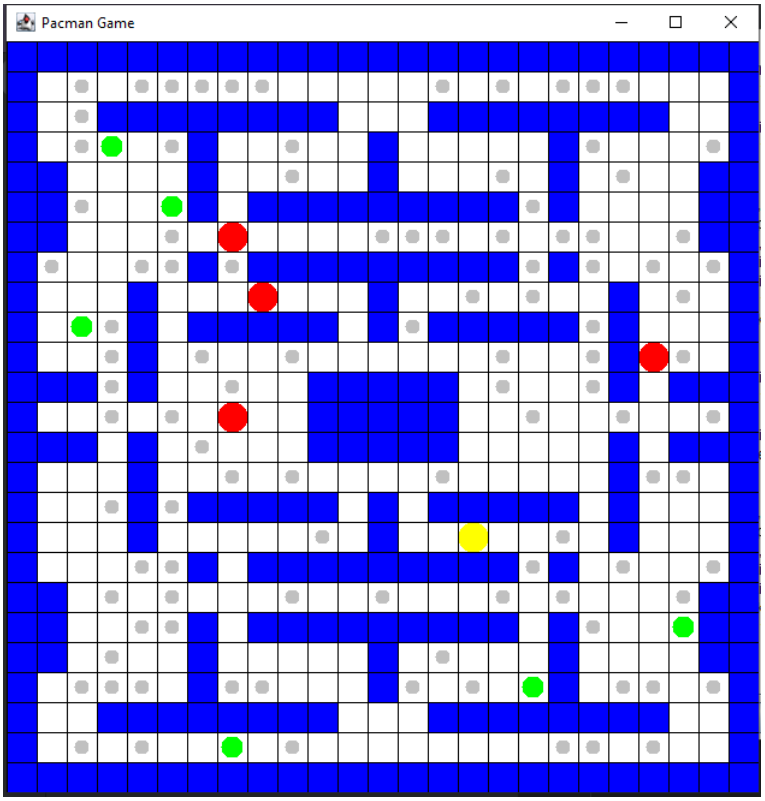

Fig. 8 - GUI with Pacman example 2.

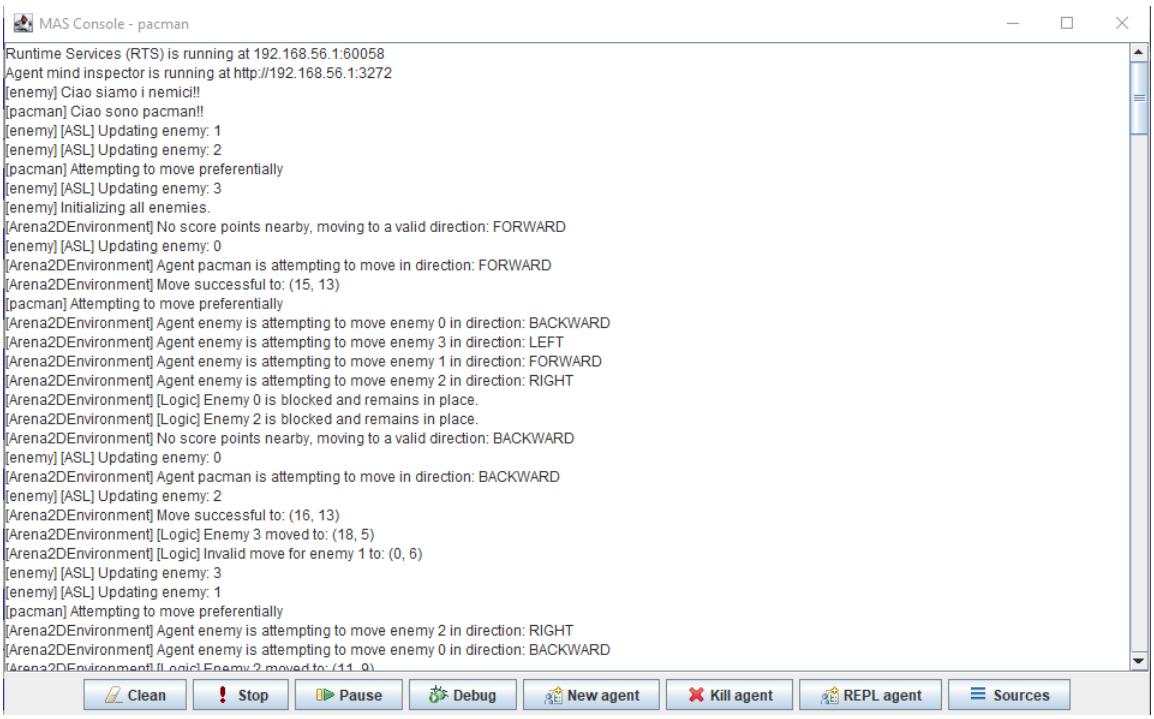

Fig. 9 - MAS that shows the logic behind the GUI, when game initialized.

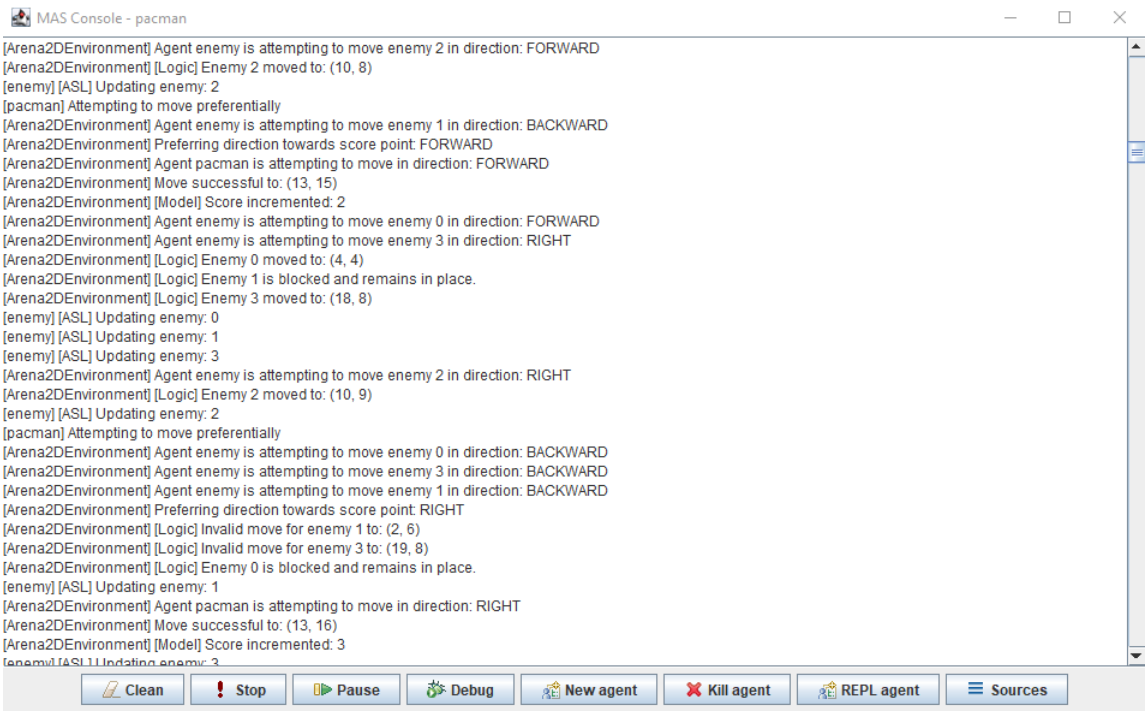

Fig. 10 - MAS that shows the logic behind the GUI, when the game is running and pacman is moving.

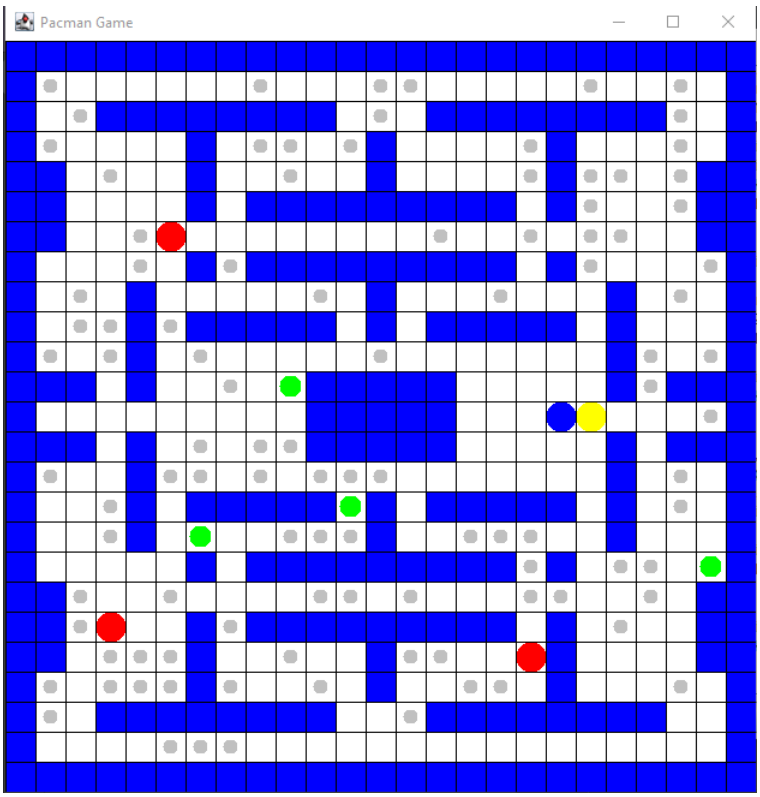

Fig. 11 - After taking a powerup Pacman has 15 seconds to kill by touching enemies and they become blue if disabled.

8 Conclusion

The Pacman project was an interesting and rewarding challenge. It allowed the application of theoretical concepts learned during the course in a practical context, improving programming and problem-solving skills. The final game is stable, enjoyable, and can serve as a foundation for further improvements and additional features.

Some additional features I would have implemented if I had more time include:

- A multiplayer pacman agent.
- Reinforcement Learning to make the pacman and the enemy learn from their mistakes.

Besides that i enjoyed this project and all the intelligent system aspect, was very fun! Thank you.