Project Report

Distributed Artificial Intelligence and Intelligent Agents

(ID2209)

KTH ROYAL INSTITUTE OF TECHNOLOGY

Group : 11

Name : Md Sakibul Islam
Name : Md Ahsanul Karim
Name : Singvalliyappa Velayutham

Date : 14 November, 2025

1. General Overview: Basic Festival with Stores

In this assignment, we made a model with several agents using the GAMA system, focusing mainly on how agents interact in a festival environment. The main goal was to copy what people do at a festival (shown as Person agents) as they deal with different spots, such as FoodStore and DrinkStore agents. We made sure each agent looked different, using different shapes and colours, and we added them to a simulation to make them interact with each other.

A big part of this work was making the SecurityGuard agent, who had to keep things safe by finding and getting rid of Person agents who have bad behaviour, based on details from the InfoCenter.

2. Running Instructions

To run the simulation model developed in this assignment, we need to follow these steps:

- Open GAMA Platform.
- Import the provided project files into GAMA by selecting File > Import, navigating to the 'General' folder, and choosing 'Existing Projects into Workspace'. Select the archive file and complete the import process.
- Locate the Main.gaml model within the imported project.
- Select the main experiment, which serves as the entry point for running the simulation.
- Press the 'Play' button seems like “Festival_Simulation” to initiate the simulation run.
- We coded all in 1 gaml file where we first develop the basic part and then we added challenge-1 & challenge-2 later on.

3. Species

Agent Person

The Person agent (Green) represents a festival guest with two changing attributes: HUNGER and THIRST. These values decrease over time. When either goes below a threshold, the Person goes to the InfoCenterPoint to request directions to an appropriate Store. After receiving the store location, the Person moves there,

replenishes the needed attribute, and then returns to normal idle behavior until it becomes hungry or thirsty again.

Agent InfoCenterPoint

The InfoCenterPoint is a stationary agent (Red) that acts as the festival's information booth. Its role is to receive requests from Person agents, identify whether the need is for food or water, locate the nearest Store that can satisfy that need, and send the store's location back to the Person.

Agent Store

The Store agent is a fixed-location service point with specific traits such as FOOD (Yellow), WATER (Blue), or both. When a Person arrives, the Store replenishes the corresponding attribute (hunger or thirst). Stores do not move, they simply wait for guests to visit.

4. Section : Basic Festival with Stores

4.1 : Explanation:

The basic part requires building a festival simulation where guests get hungry or thirsty, ask the InfoCenterPoint for directions, and then go to the correct Store to refill their needs. All three agent types must be included, visually distinct, and able to interact properly.
We solved it like, there are 3 stores : Food , drink and both . The agents have a chance of 2% to either become hungry , thirsty or both . If they don't go through any of these , they wander . Once they are in need , they go to the information centre and once they reach there they get the location of the store that has their need . A reflex checks their need at every cycle and another reflex checks if they have reached the information store.

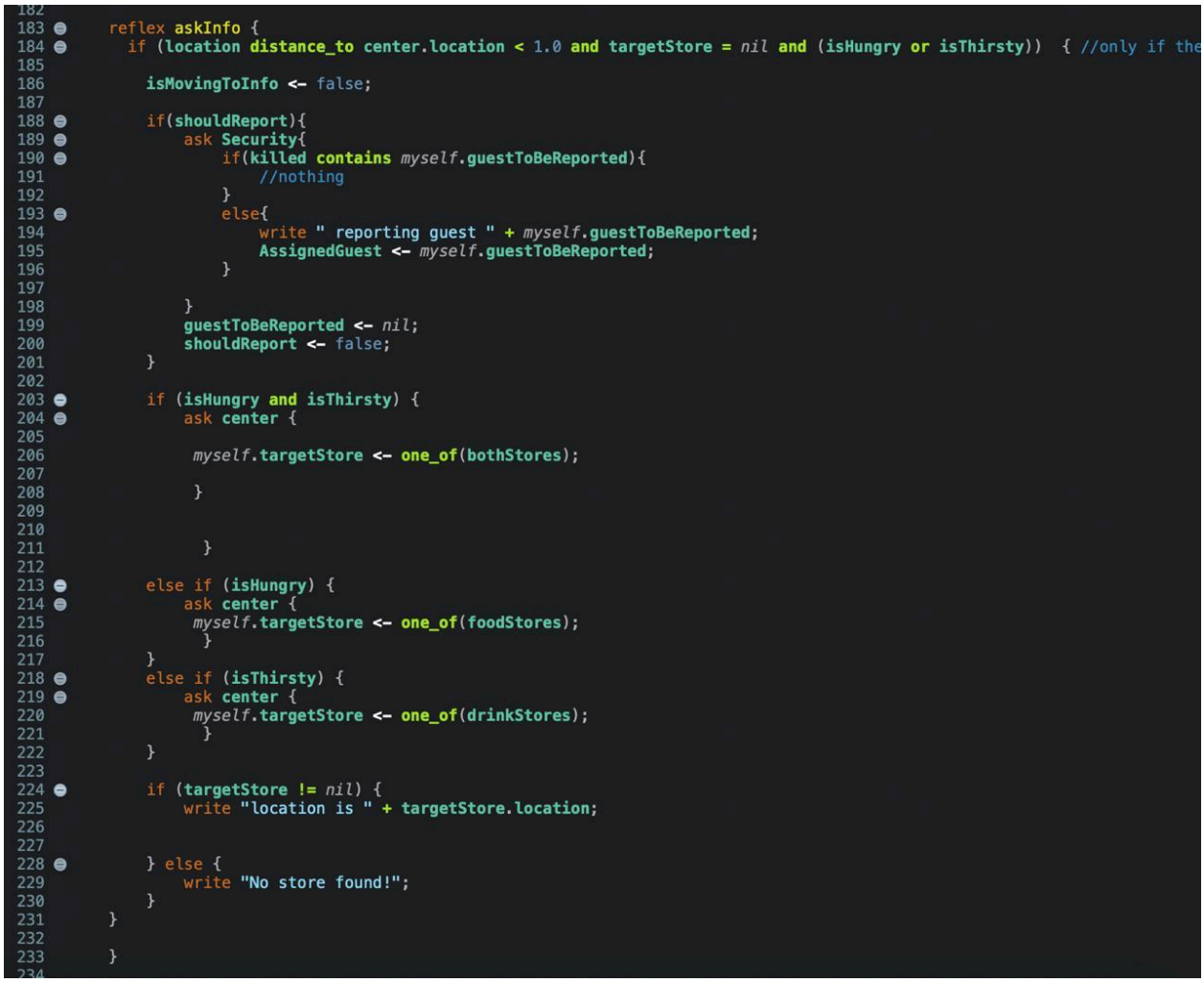
4.2 : Code:

- **Relevant Code Snippet(s) :**

```
reflex changeState {
    if (!isHungry) {
        isHungry <- flip(0.02);
    }
    if (!isThirsty) {
        isThirsty <- flip(0.01);
```

```
        }
        if(!isHungry and !isThirsty and !evil ){
            evil <- flip(0.0005);
            if(evil){
                    write "turned evil";
            }
        }
    }
```

- **Code Screenshots:**



```
182
183     reflex askInfo {
184       if (location distance_to center.location < 1.0 and targetStore = nil and (isHungry or isThirsty))  { //only if the
185
186         isMovingToInfo <- false;
187
188         if(shouldReport){
189             ask Security{
190                 if(killed contains myself.guestToBeReported){
191                     //nothing
192                 }
193                 else{
194                     write " reporting guest " + myself.guestToBeReported;
195                     AssignedGuest <- myself.guestToBeReported;
196                 }
197
198             }
199             guestToBeReported <- nil;
200             shouldReport <- false;
201         }
202
203         if (isHungry and isThirsty) {
204             ask center {
205
206              myself.targetStore <- one_of(bothStores);
207
208              }
209
210
211               }
212
213         else if (isHungry) {
214             ask center {
215              myself.targetStore <- one_of(foodStores);
216               }
217         }
218         else if (isThirsty) {
219             ask center {
220              myself.targetStore <- one_of(drinkStores);
221               }
222         }
223
224         if (targetStore != nil) {
225             write "location is " + targetStore.location;
226
227
228         } else {
229             write "No store found!";
230         }
231     }
232
233     }
234
```

```
    reflex goToStore when: (targetStore != nil) {
        do goto target: targetStore.location;
    }


    reflex checkArrivalAtStore{
        if(targetStore != nil ){
            if (location distance_to targetStore.location < 1.0) {
              do addToStore(targetStore,getNeedType());
              write "Guest arrived at store!";
              isHungry <- false;
              isThirsty <- false;
              targetStore <- nil;
            }

        }
    }
```

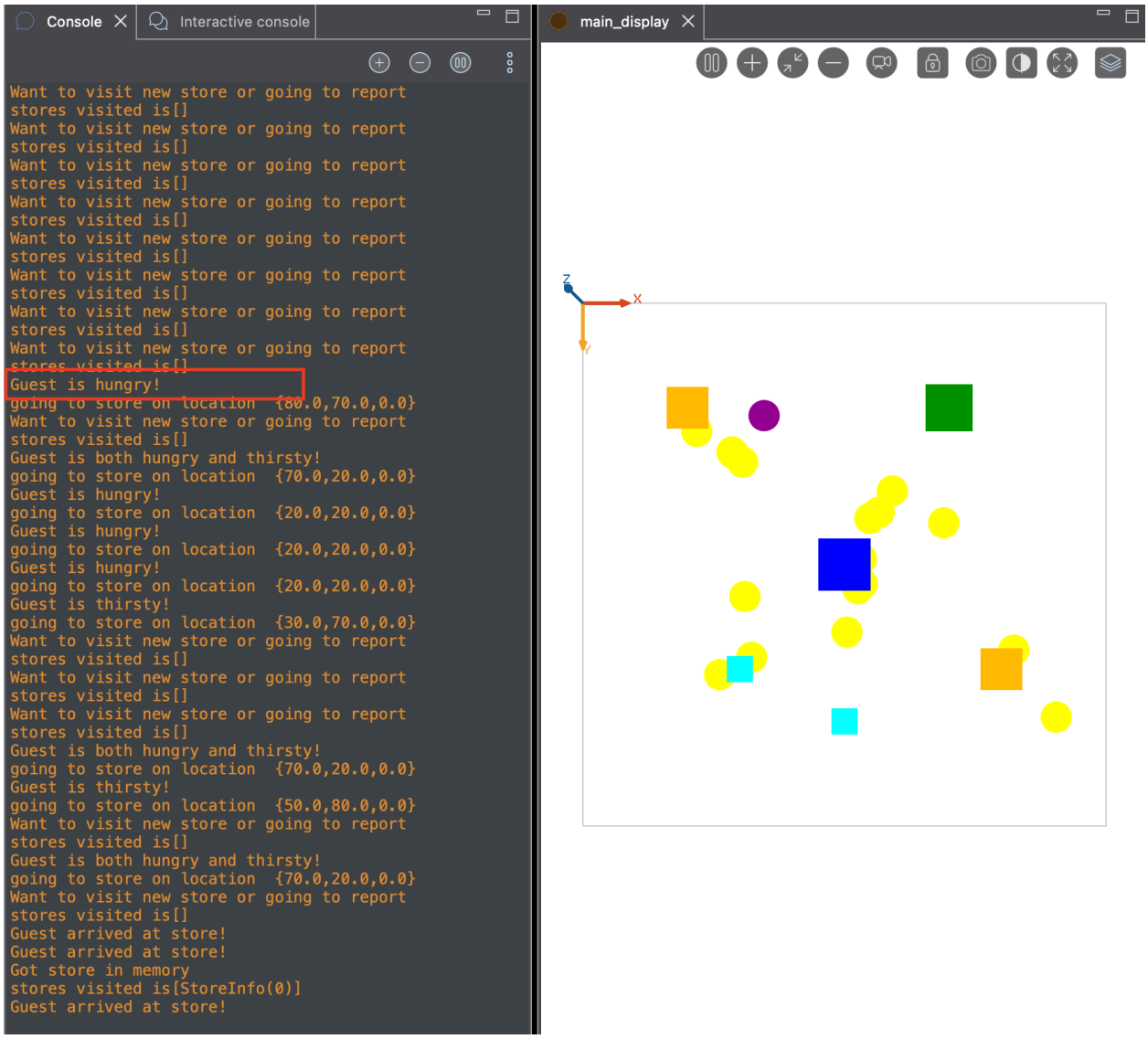
```
reflex changeState {
        if (!isHungry) {
            isHungry <- flip(0.02);
        }
        if (!isThirsty) {
            isThirsty <- flip(0.01);
        }
        if(!isHungry and !isThirsty and !evil ){
            evil <- flip(0.0005);
            if(evil){
                write "turned evil";
            }
        }
    }
```

- **Explanation of the code :**
  The code controls how a guest behaves in the simulation. When the guest reaches the information center and needs help, the correct type of store based on hunger or thirst. After receiving a target store, the guest moves toward it, and once close enough, it resets its hunger or thirst and clears the target. Meanwhile, the guest's state updates over time: it can randomly become hungry, thirsty.

4.3 : Demonstration

- **Use Case - 1 : Agent Person (Guest) is Hungry**
  - **Description :** When the Guest becomes hungry, it goes to the InfoCenter, asks for a food store, receives the store location, travels there, and restores its hunger level before returning to normal activity.

- **Screenshot of program execution/output :**

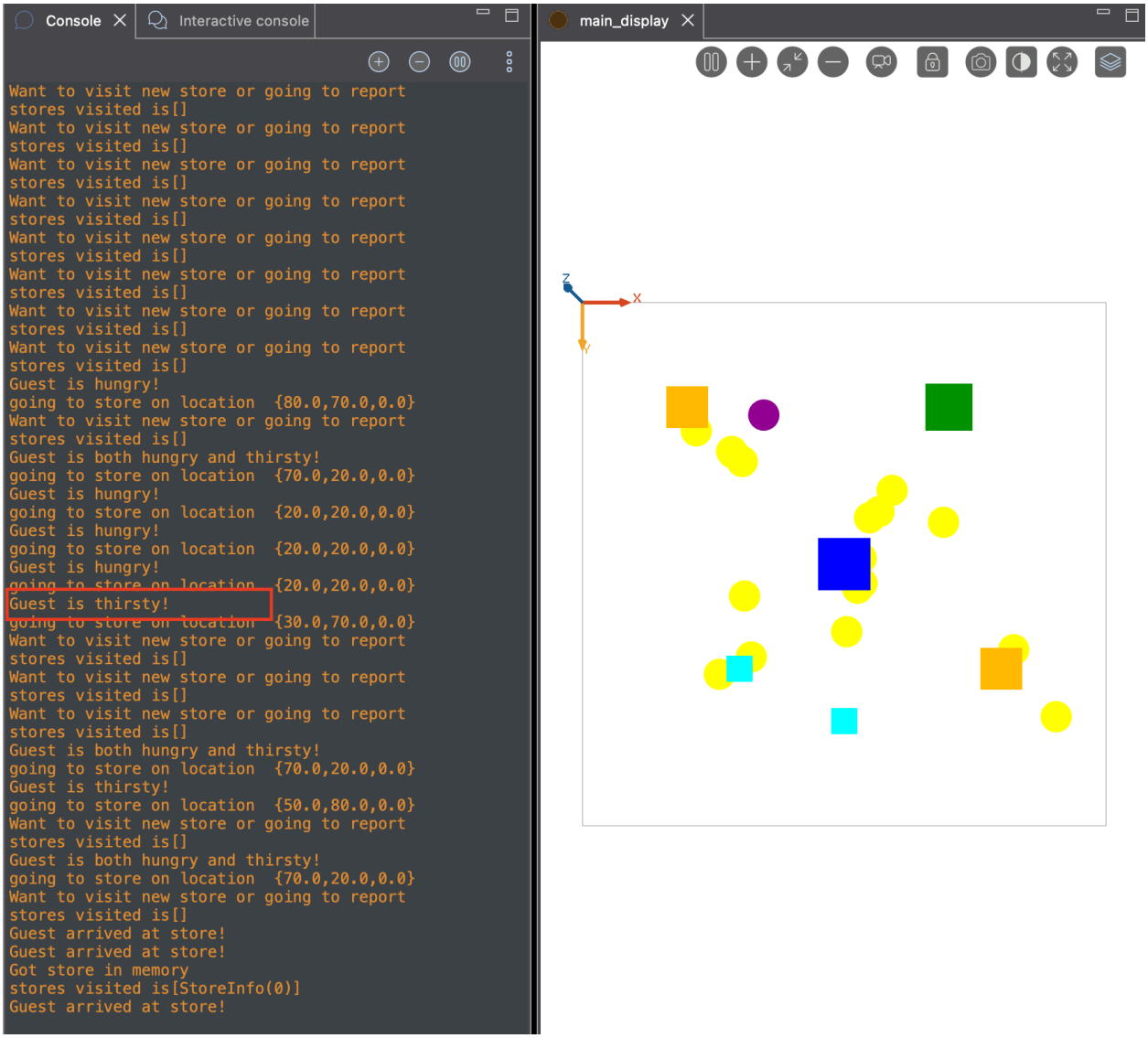


- **Short interpretation of the result :**
  The output shows that Guests successfully request store locations, travel to different stores, and arrive there as expected. The logs confirm that store visits are recorded and the movement between InfoCenter and stores behaves consistently. Visually, the Guests move around the environment, reach stores, and continue cycling through their needs as intended.

- **Use Case - 2 : Agent Person (Guest) is Thirsty**
  - **Description :** The Guest's thirst level drops below its threshold, marking the agent as thirsty. This triggers the behaviour where the Guest must

seek water by going to the InfoCenter and requesting the location of a drink-providing store.

- **Screenshot of program execution/output :**



- **Short interpretation of the result :** The output shows when the Guest becomes thirsty, it correctly identifies the need, requests a drink store, receives the appropriate store location, and begins moving toward it. The logged messages confirm that the thirst-related logic and store selection are functioning as expected.

5. Challenge-1: Memory of Agents - Small Brain

5.1 : Explanation:

This challenge part requires a simple memory system to the Guest so it can remember previously visited stores and use that information when choosing where to go. We implemented this by keeping a list of visited stores inside each Guest. Whenever a new store is visited, it gets added to memory. During decision-making, the Guest checks this memory first and based on the randomness we decide whether we go to a known store or choose a new one.

5.2 : Code:

- **Relevant Code Snippet(s) :**

```
if ((isHungry or isThirsty) and targetStore = nil and !isMovingToInfo) {
        float randomValue <- rnd(1.0);
                        if (randomValue < 0.1 or length(visited) <= 0 or shouldReport) { //
discover new place
                                write "Want to visit new store or going to report";
                                 isMovingToInfo <- true;
                                }
                        else{
                                Store s <- getStore(getNeedType());
                                if(s != nil){
                                        write "Got store in memory";
                                        targetStore <- s;
                                }
                                else{
                                        write "Needed store not in memory";
                                        isMovingToInfo <- true;
                                }
                        }
                        write "stores visited is" + visited;
    }
    else if (!isHungry and !isThirsty and targetStore = nil) {
       do wander;
    }
```

- **Code Screenshots:**

```
action addToStore(Store s,string t){

        create StoreInfo{
            store <- s;
            type <- t;
        }
       add item:last(StoreInfo) to: visited;

}

action getStore(string t) {
StoreInfo match <- visited first_with (each.type = t);
if (match != nil) {
    return match.store;
}
return nil;
}
```

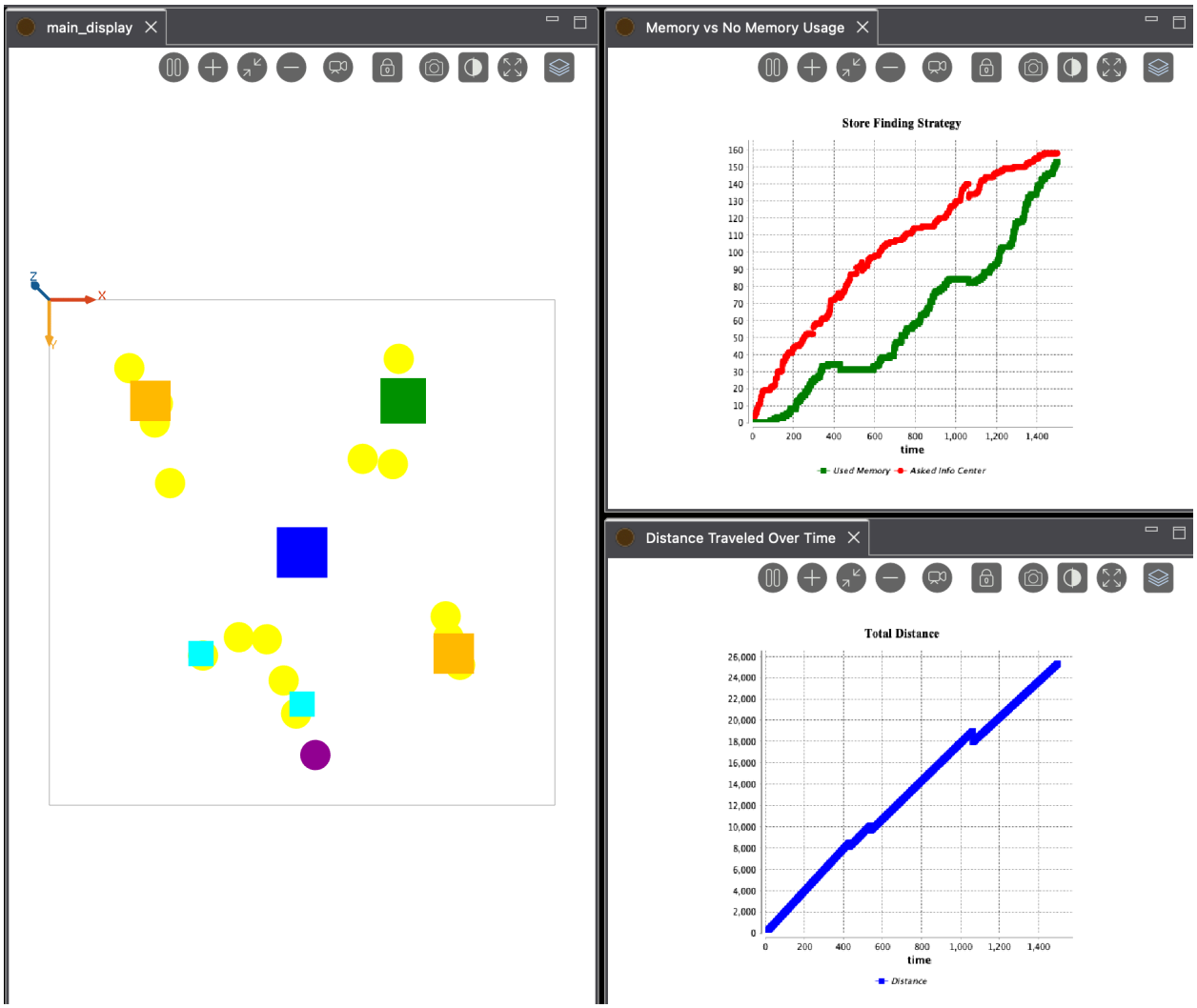
```
    float randomValue <- rnd(1.0);
    if (randomValue < 0.1 or length(visited) <= 0 or shouldReport) { // discover new place
        write "Want to visit new store or going to report";
         isMovingToInfo <- true;
        }
```

```
            Store s <- getStore(getNeedType());
            if(s != nil){
                write "Got store in memory";
                targetStore <- s;
            }
```

```
            else{
                write "Needed store not in memory";
                isMovingToInfo <- true;
            }
```

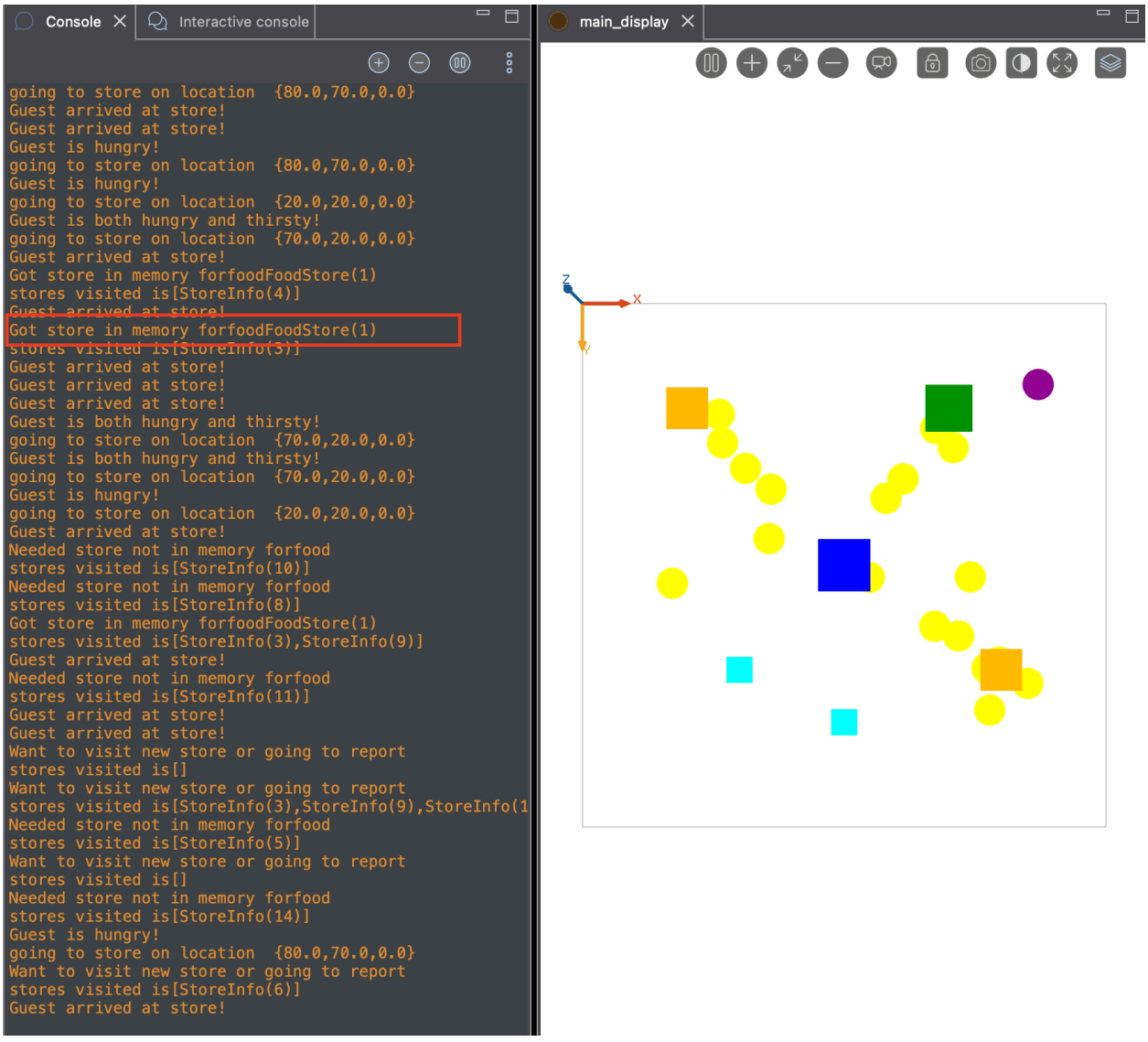
- **Explanation of the code :** The “addToStore” action saves a visited store into the Guest’s memory by creating a “StoreInfo” object (containing the store and its type) and adding it to the visited list. The “getStore” action checks this memory to find a previously visited store that matches the needed type. If found, it returns that store, otherwise it returns “nil”. The main logic uses this memory: if a matching store is found, the Guest prints “Got store in memory” and sets it as “targetStore”. If not, it prints “Needed store not in memory” and moves toward the InfoCenter to ask for a new store. Additionally, a random check allows the Guest to sometimes explore new stores even if memory exists, making the behaviour more natural.

- **Comparing distance traveled :**



5.3 : Demonstration:

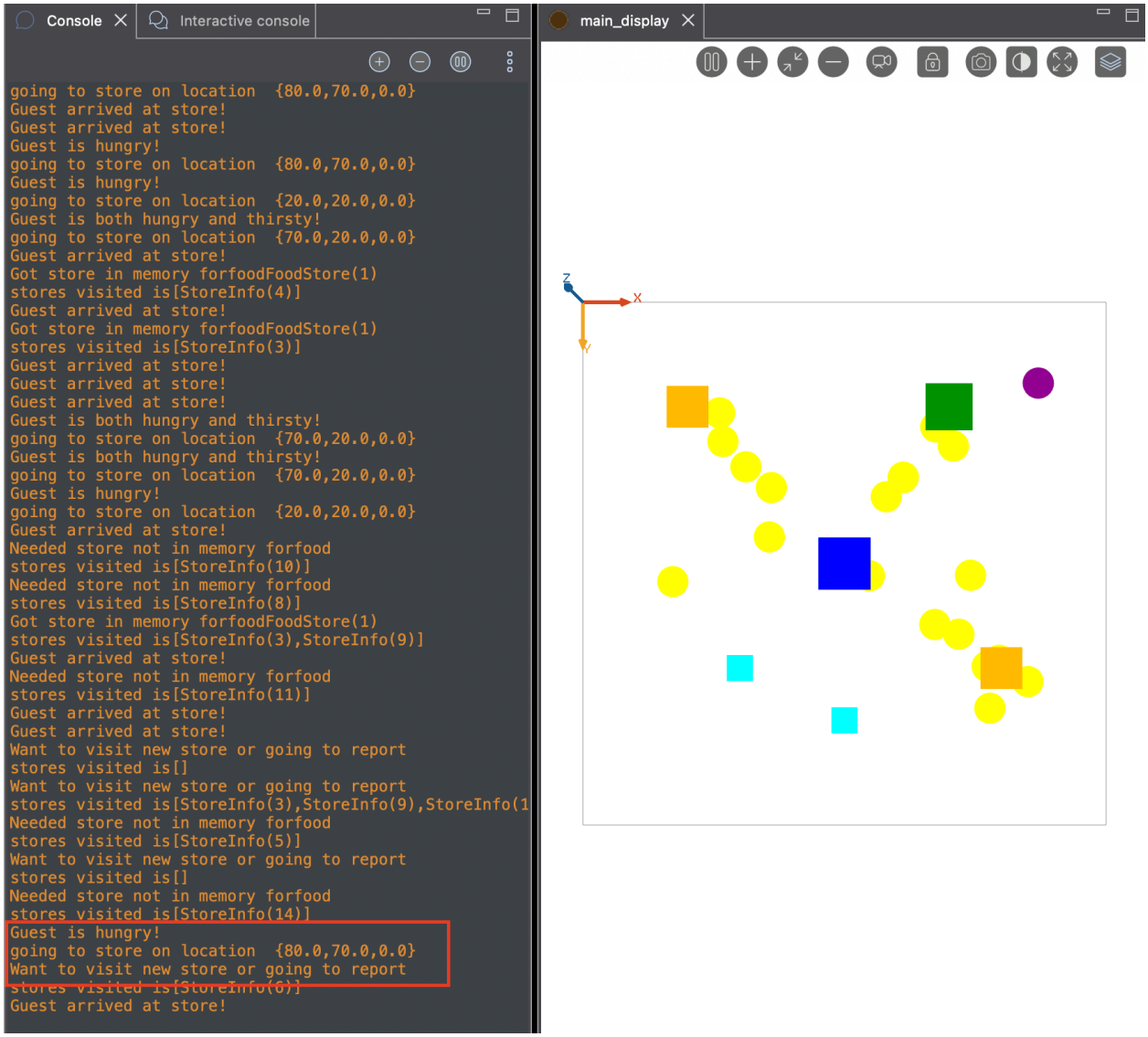
- **Use Case - 1 : Guest Visits a Known Store**
    - **Description :** The Guest finds a matching store in its memory and directly goes there without asking the InfoCenter, making the trip quicker and more efficient.
    - **Screenshot of program execution/output :**



    - **Short explanation of the results:**
      The Guest successfully found a matching store in its memory and used it immediately. This shows that the memory system is working, allowing the

Guest to choose a known store quickly without needing help from the InfoCenter.

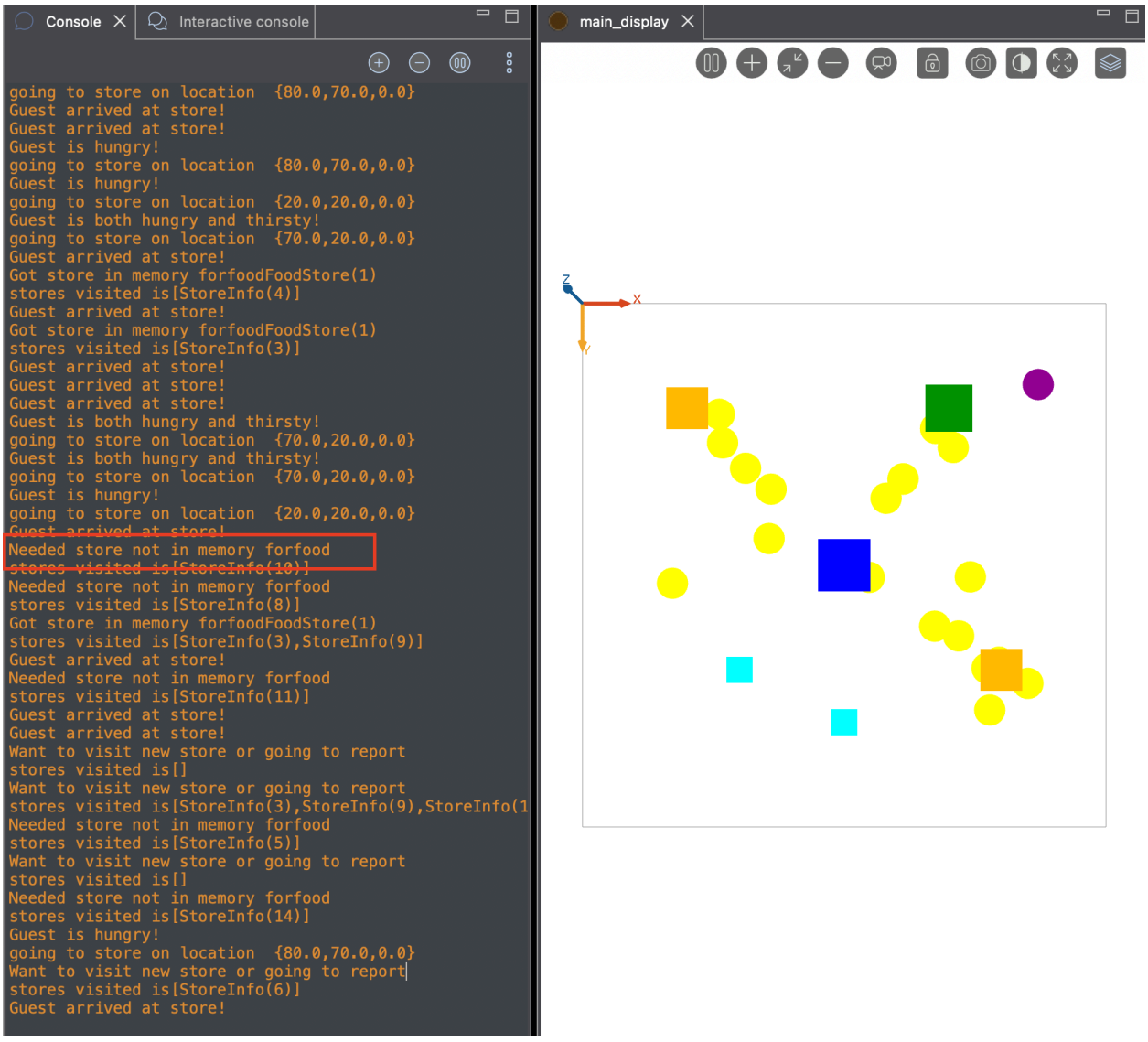
- **Use Case - 2 : Guest Tries a New Store**
    - **Description :** When the Guest needs food or water but decides not to use a previously visited store, it chooses a new store instead. This happens either because its memory is empty or because the random choice encourages exploration. The Guest then moves toward this new store, visits it, and adds it to memory for future use.
    - **Screenshot of program execution/output :**



- **Short explanation of the results:** The Guest choosing a new store location instead of relying on memory. This confirms that the exploration logic works, allowing the Guest to visit unfamiliar stores and expand its memory list.

- **Use Case - 3 : Needed Store Not in Memory**

  - **Description :** When the Guest becomes hungry or thirsty and checks its memory, it finds that no previously visited store matches the needed type. Since the required store isn't in memory, the Guest decides to go to the InfoCenter to request a new store location and then continues its journey based on the received information.
  - **Screenshot of program execution/output :**

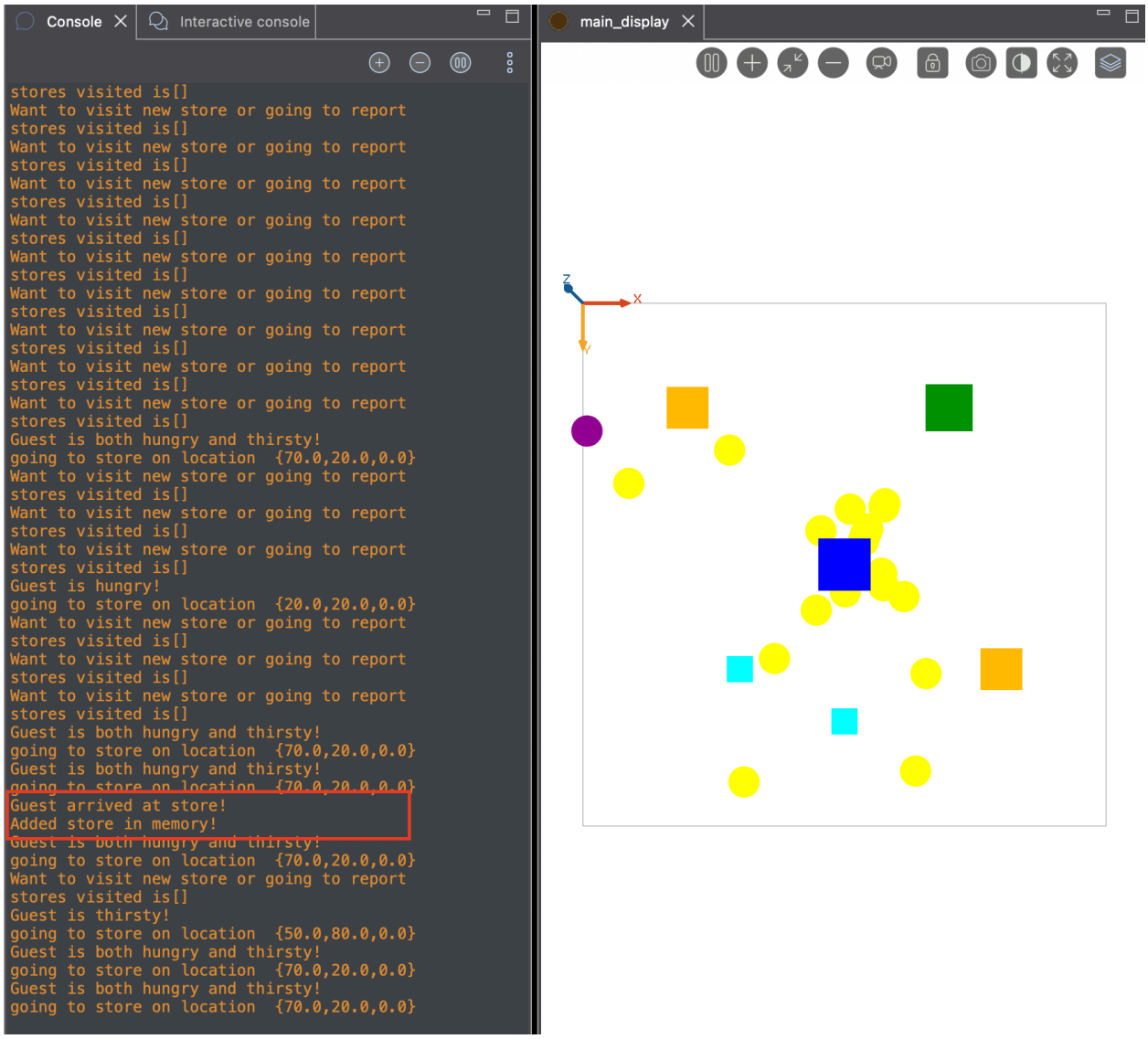


- **Short explanation of the results:** The Guest couldn't find a matching store in its memory and therefore triggered the fallback behavior of going to the InfoCenter. This confirms that the memory check works correctly and the Guest only relies on stored locations when they exist.

- **Use Case - 4 : Guest Updates Memory After Visiting a New Store**

  - **Description :** After the Guest visits a new store for the first time, it saves that store's information into its memory list. This updated memory helps the Guest easily choose the same store in future needs without asking the InfoCenter again, making future decisions faster and more efficient.

  - **Screenshot of program execution/output :**



- **Short explanation of the results:** After reaching a new store, the Guest successfully adds it to memory. This shows the memory update system is working, allowing the Guest to reuse this store in future needs.

6. Challenge 2: Removing Bad Behaviour Agents

6.1 : Explanation:

This challenge is about detecting badly behaving guests and removing them from the festival. In our solution, a Guest can randomly become “evil”, and other Guests can decide to report this bad Guest to the InfoCenter. The InfoCenter stores the reported Guest and calls a Security Guard agent, passing the target to it. The Security Guard then moves to the reported Guest and, once close enough, uses the die function to remove that Guest from the simulation.

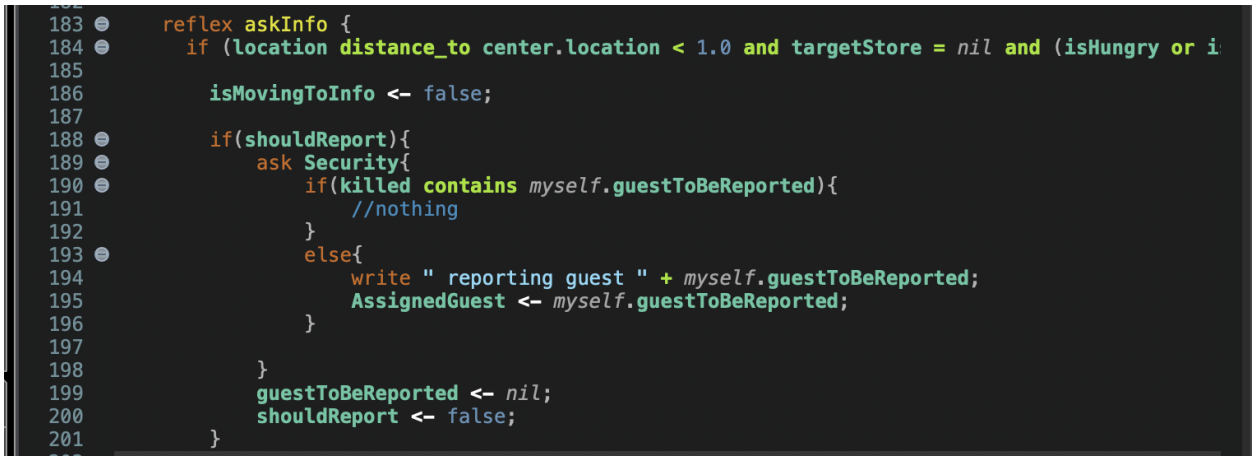
6.2 : Code:

- **Relevant Code Snippet(s) :**

```
species Security skills:[moving]{

        Guest AssignedGuest <- nil;
        list<Guest>  killed <- [];


        aspect base {
     draw circle(3) color: #purple;
  }

  reflex GotoGuest when : (AssignedGuest !=nil){
          do goto target: AssignedGuest.location;

  }

  action addToKillList(Guest g){
        add item:g to: killed;
  }


  reflex checkForProblem{
```

```
        if(AssignedGuest != nil){
                if(!(killed contains AssignedGuest)){
                        write "kill list " + killed;
                if (location distance_to AssignedGuest.location < 1.0 ) {
                        write "killing guest ";
                        do addToKillList(AssignedGuest);
                        ask AssignedGuest{
                                do die;
                        }
                        AssignedGuest <- nil;
                }
        }

        }
        else{
                do wander;
        }
    }
}
```

- **Code Screenshots:**



```
183     reflex askInfo {
184       if (location distance_to center.location < 1.0 and targetStore = nil and (isHungry or i
185
186         isMovingToInfo <- false;
187
188         if(shouldReport){
189             ask Security{
190                 if(killed contains myself.guestToBeReported){
191                     //nothing
192                 }
193                 else{
194                     write " reporting guest " + myself.guestToBeReported;
195                     AssignedGuest <- myself.guestToBeReported;
196                 }
197
198             }
199             guestToBeReported <- nil;
200             shouldReport <- false;
201         }
```

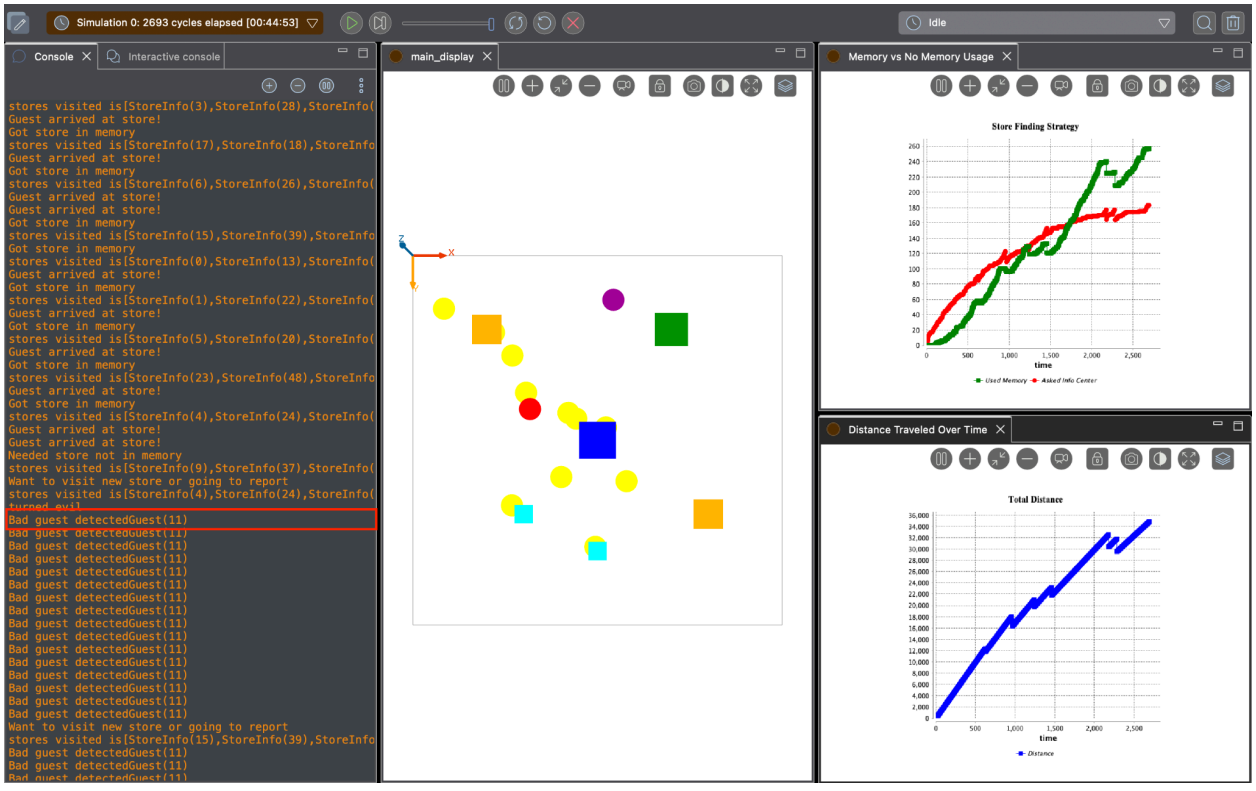
```
138
139     reflex checkForBadGuests{
140         list<Guest> evil_guests <- (Guest - self) where (each.evil);
141
142         // 2. if the list is empty → return nil
143         if (!empty(evil_guests)) {
144            guestToBeReported <- evil_guests closest_to(self);
145            shouldReport <- true;
146         }
147
148     }
149
```

- **Explanation of the code :** The checkForBadGuests reflex searches for all nearby Guests marked as evil and selects the closest one. If such a Guest exists, it sets guestToBeReported and enables shouldReport, meaning this agent will report the bad Guest. In the askInfo reflex, when the reporting Guest reaches the InfoCenter, it checks if reporting is needed. If yes, it sends the report to the Security agent. If the Guest hasn't already been removed, the InfoCenter assigns this bad Guest as the Security Guard's target (AssignedGuest). After reporting, the flags are reset. This flow allows Guests to detect bad behaviour, report it, and trigger the Security Guard to take action.

6.3 : Demonstration:

- **Use Case - 1 : Guest Detects a Bad Guest**
  - **Description :** A normal Guest notices that another Guest has turned evil. It identifies this bad Guest as the closest evil agent and marks it for reporting.

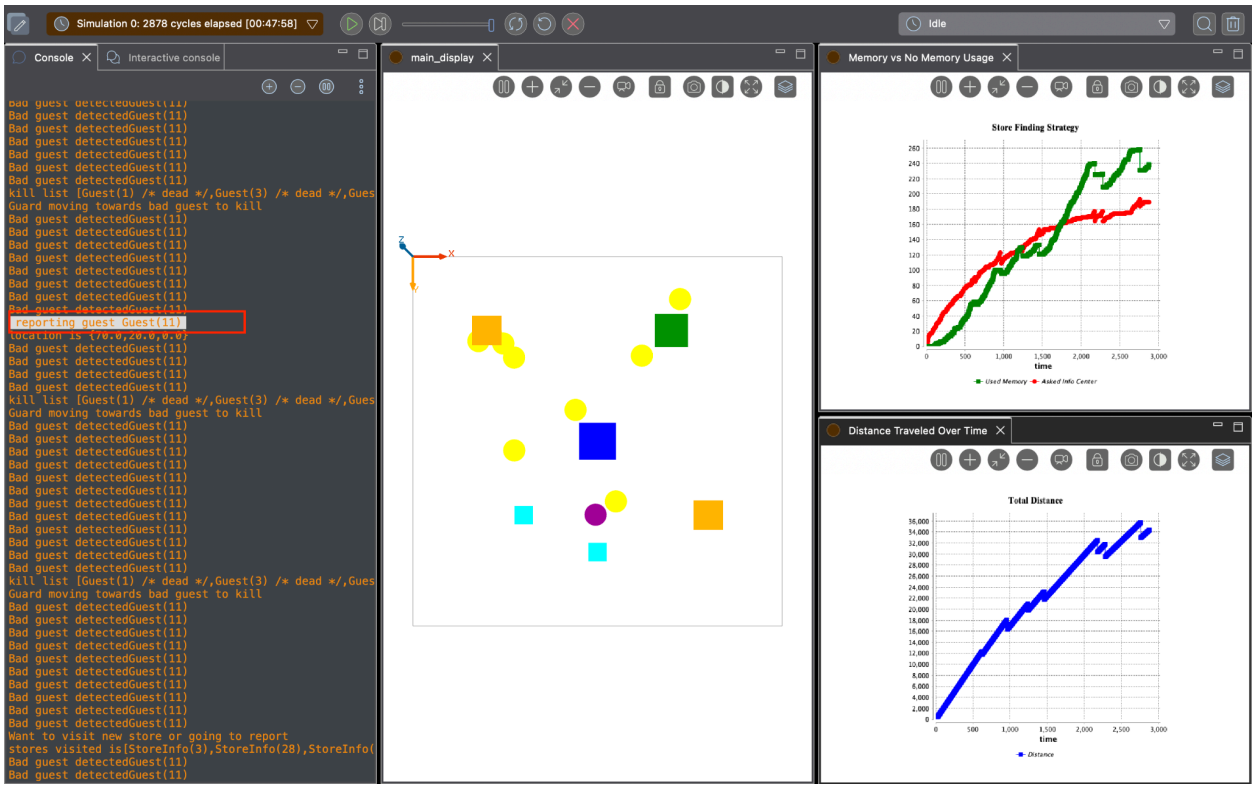
  - **Screenshot of program execution/output :**



  - **Short explanation of the results:** It shows "Bad guest detected" along with the Guest ID, confirming that the system correctly identifies when a Guest has turned evil. This means the detection logic is working, and

normal Guests are successfully spotting bad behaviour inside the simulation.

- **Use Case - 2 : Guest Reports to the InfoCenter**
    - **Description :** Once the reporting Guest reaches the InfoCenter, it sends the information about the evil Guest. The InfoCenter receives the report and assigns that bad Guest as the target for the Security Guard.
    - **Screenshot of program execution/output :**

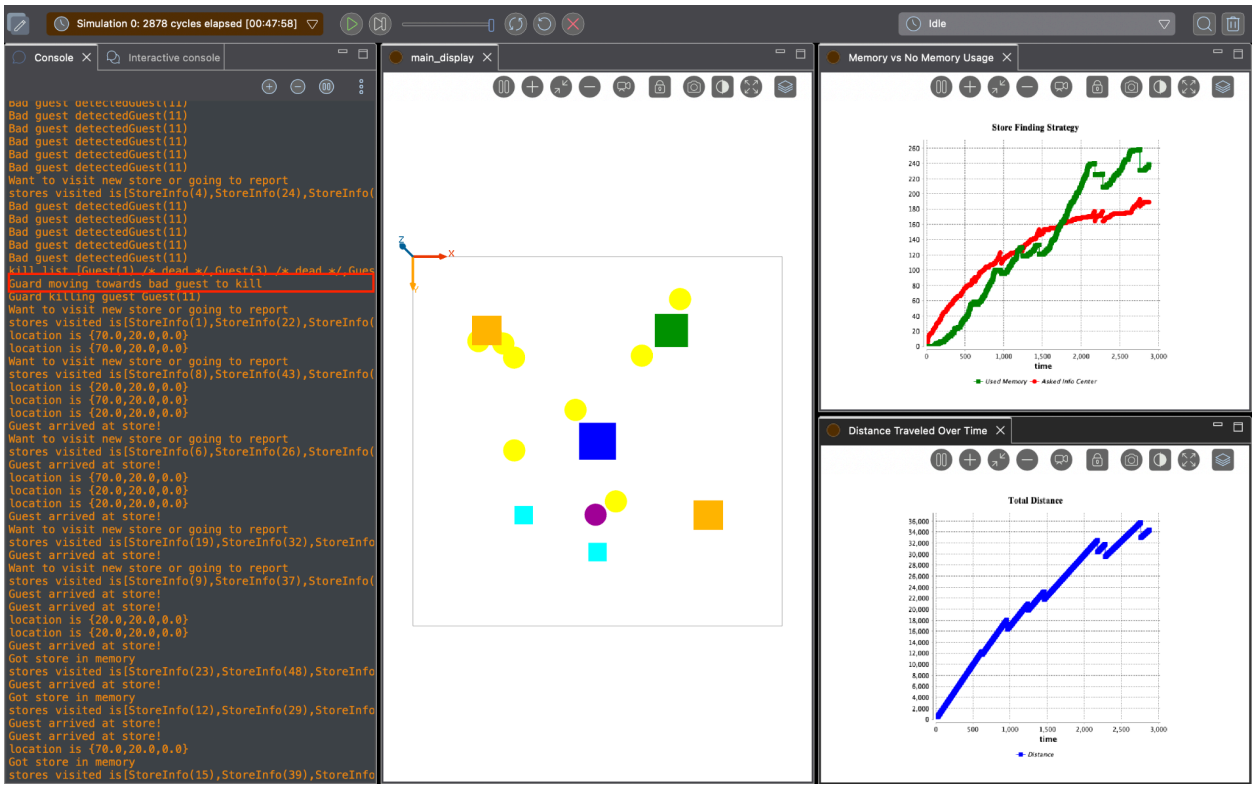


    - **Short explanation of the results:** A Guest successfully reported the bad Guest to the InfoCenter, confirming that the reporting mechanism works. The system correctly logs the reporting event and provides the bad Guest's ID, showing that communication between the Guest and InfoCenter is functioning as intended.

- **Use Case - 3 : Security Guard Moves Toward the Bad Guest**

- **Description :** After receiving the assignment, the Security Guard starts moving directly toward the location of the reported bad Guest to handle the situation.

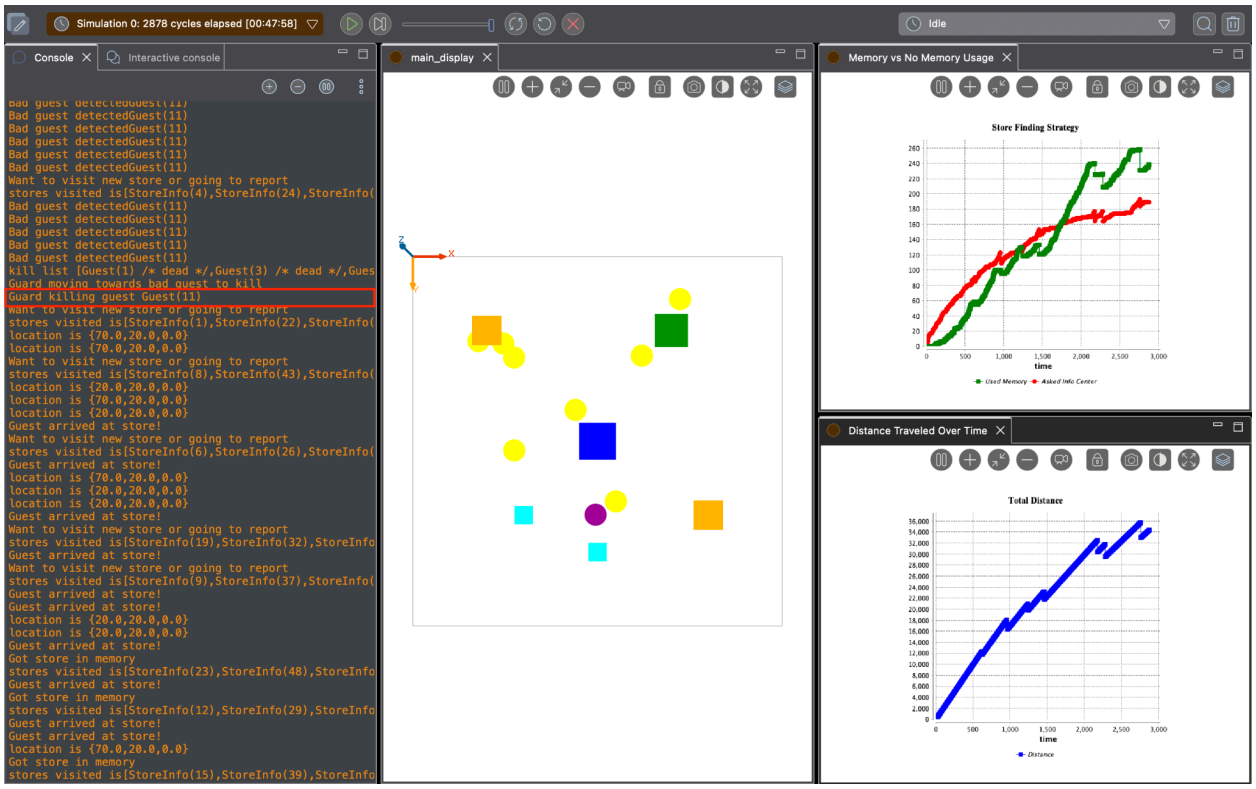
- **Screenshot of program execution/output :**



- **Short explanation of the results:** The Security Guard moving toward the bad Guest and then executing the kill action (“Guard killing guest”). This confirms that the reporting system worked, the Guard received the correct target, and the removal of the bad Guest happened successfully inside the simulation.

- **Use Case - 4 : Security Guard Removes the Bad Guest**
  - **Description :** When the Security Guard reaches the evil Guest, it executes the die function to remove that Guest from the environment, successfully resolving the disturbance.

- **Screenshot of program execution/output :**



- **Short explanation of the results:** The Security Guard reached the reported bad Guest and executed the kill command. This shows that the full workflow detection, reporting, assignment, and removal has completed successfully, proving the Guard's elimination logic works as intended.

7. Final Remarks :

- **Summarize :** Through this assignment, we learned how to design interacting agents in GAMA, implement communication between them, manage internal states like hunger/thirst, and add memory-based decision-making for more realistic behaviour. I also gained experience in detecting abnormal behaviour, reporting it, and coordinating a separate agent (Security Guard) to act on that information.

- **Limitations or Improvements :** A few limitations remain like the behaviour logic can be expanded for more complex decision-making, the memory system could include forgetting or prioritisation, and the Security Guard could handle multiple reports more efficiently. Future improvements could also include smoother movement, better visualisation, and more diverse types of bad behaviour.