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
Assignment : 03
Date : 28 November, 2025

1. General Overview:

This assignment has multi agent cooperation and decision making across two main tasks and one challenge. In the first task, each queen in the N Queen problem acts as an independent agent that communicates only with the queen before and after it to find positions on the board without conflicts. The second task expands the festival simulation by adding stage agents with different attributes and guest agents that use FIPA communication to ask for stage details, calculate utility values, and choose the best option. The challenge then focuses on global utility, where guests share their choices, consider crowd preferences, and adjust their decisions together so that the overall utility for all agents is as high as possible.

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 Task-1_Queen.gaml model within the imported project.
- Locate the Task2.gaml model within the imported project.
- Locate the Challenge1.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 “NQueens” to initiate the simulation run for Task-1.
- Press the 'Play' button seems like “Festival_Simulation” to initiate the simulation run for Task-2.
- Press the 'Play' button seems like “Festival_Simulation” to initiate the simulation run for Challenge-1. Please kindly play a bit slow.
- We coded in 3 gaml file where we first developed the task-1 and then we added task-2 & challenge-1 later on.

3. Species

Agent Queen

The Queen species represents each queen as an autonomous agent responsible for finding a valid position on the chessboard without conflicting with other queens. Each

queen knows its own row and explores possible column positions while checking for conflicts with previously placed queens. Communication is limited to messages exchanged with its predecessor and successor, allowing queens to coordinate their placement. When a queen runs out of valid positions, it informs its predecessor and requests repositioning, which triggers a backtracking chain until a conflict-free configuration is found. Through this decentralized communication and decision making, the Queen species collectively constructs a valid N Queen solution.

Agent Stage

The Stage species represents each performance stage in the festival, with its own attribute values for light show quality, speaker strength and music style. These attributes are randomly assigned to create variation between stages. Stages use FIPA communication to respond to guest queries by sending back their attribute information. Each stage also keeps track of how many guests are currently visiting, updating its visitor count as guests arrive and leave. Visually, each stage is displayed as a coloured square, making it easy to see their positions and popularity within the festival environment.

4. Section : N Queen Problem

4.1 : Explanation:

For this task, we were asked to solve the N Queen problem using a fully agent-based approach where each queen behaves as an autonomous agent. The challenge was to place N queens on an N×N board so that none of them share the same column or diagonal, and to achieve this without centralized control.

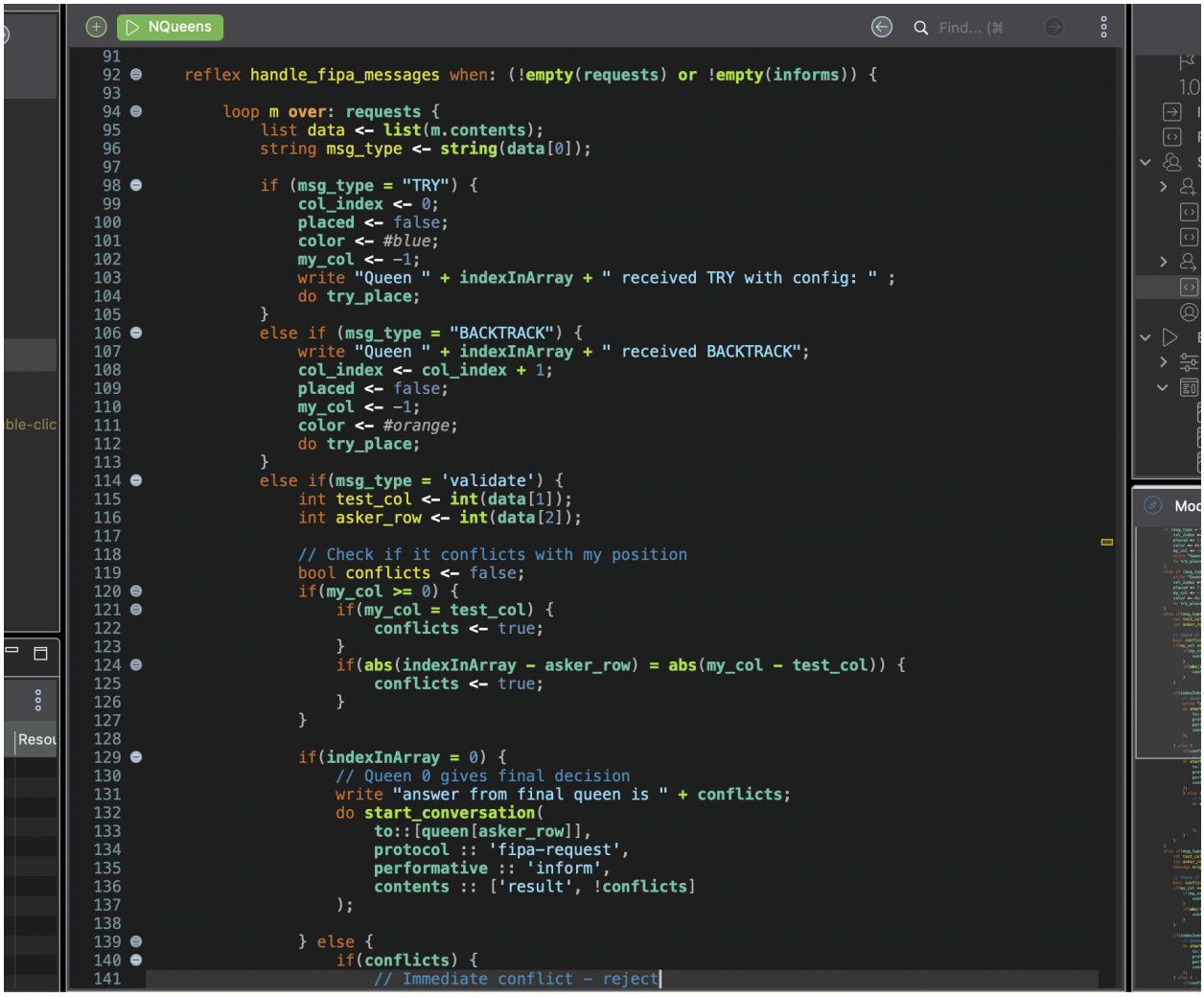
To solve it, we structured the system so that each queen is responsible for finding its own valid column while communicating only with its immediate predecessor and successor. Each queen tries its possible columns one by one and sends FIPA validation messages backward to check whether a position conflicts with previously placed queens. If a position is valid, the queen places itself and signals the next queen to start. If no valid position remains, the queen sends a backtrack request to its predecessor so the previous queen can adjust its placement. Through this distributed backtracking process, the queens collectively build a complete non-conflicting configuration, allowing us to find N-Queen solutions using only local communication between agents.

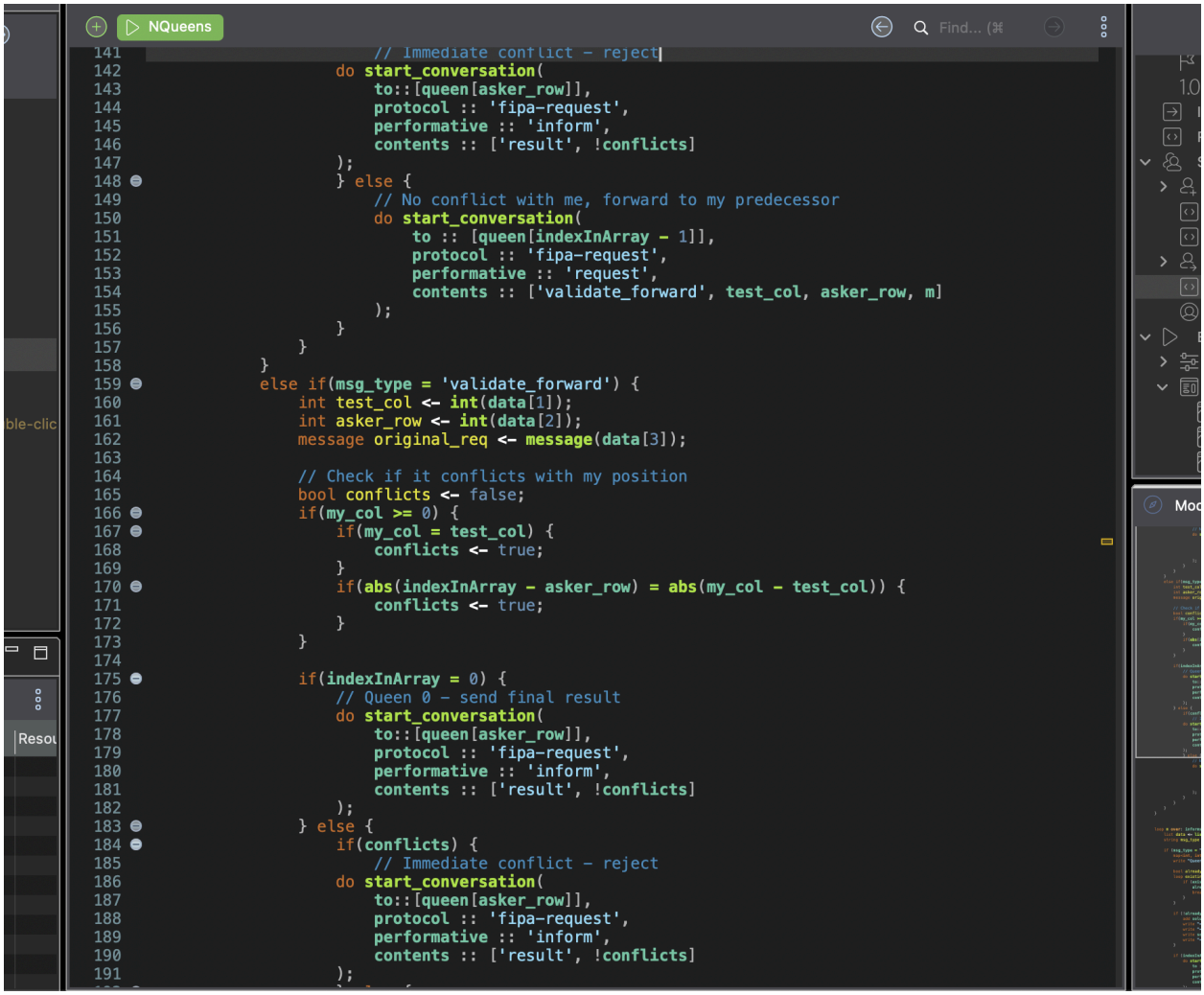
4.2 : Code:

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

```
action try_place {
   int test_col <- possible_columns[col_index];
   do checkForPredecessor(test_col);
}

action checkForPredecessor(int test_col) {
   if(indexInArray = 0){
      do start_conversation(
         to :: [self],
         protocol :: 'fipa-request',
         performative :: 'inform',
         contents :: ['result', true]
      );
   } else {
      do start_conversation(
         to :: [queen[indexInArray - 1]],
         protocol :: 'fipa-request',
         performative :: 'request',
         contents :: ['validate', test_col, indexInArray]
      );
   }
}
```

- **Code Screenshots:**



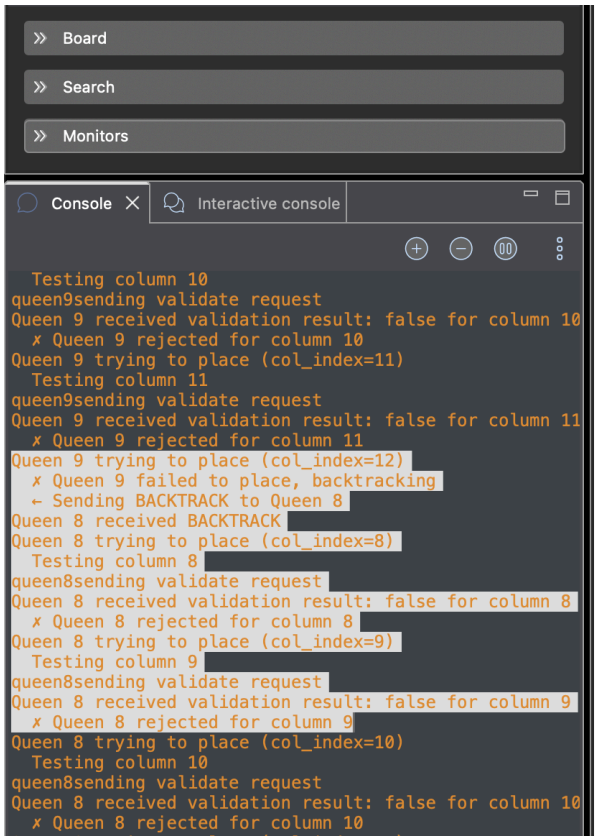


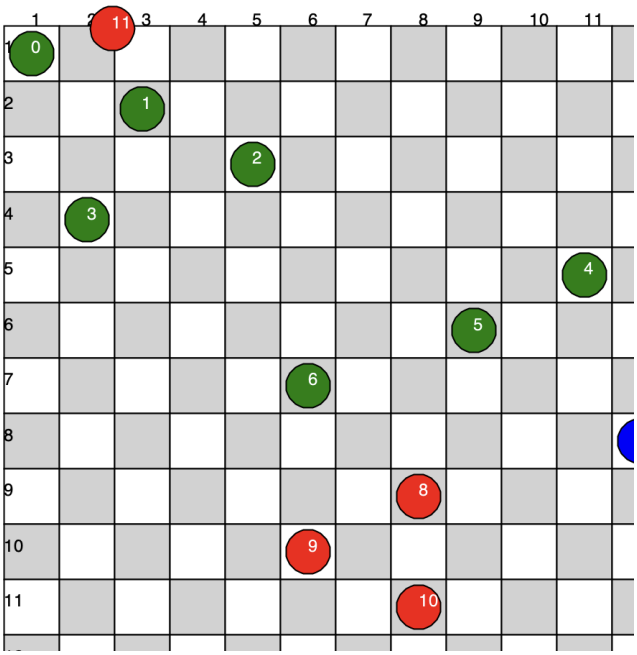
- **Explanation of the code :** In this section we handle all incoming FIPA messages for each queen. When a queen receives a TRY message, it resets its state and starts testing columns. A BACKTRACK message tells the queen to move to the next column and try again. The validate and validate_forward messages implement the distributed conflict-checking logic: each queen checks whether a proposed column conflicts with its own position and either rejects it or forwards the validation request further back to the previous queen. Queen 0 always

provides the final validation result. This messaging process allows the queens to coordinate placement, detect conflicts, and drive backtracking without any central controller.

4.3 : Demonstration

- **Use Case - 1 : Conflict & Backtracking**
    - **Description :** A queen reaches a point where all remaining column options create conflicts with previously placed queens. When no valid position is found, it sends a backtracking request to its predecessor. The previous queen then moves to its next available column and restarts the placement process. This coordinated adjustment continues until the queens collectively reach a valid arrangement.
    - **Screenshot of program execution/output :**

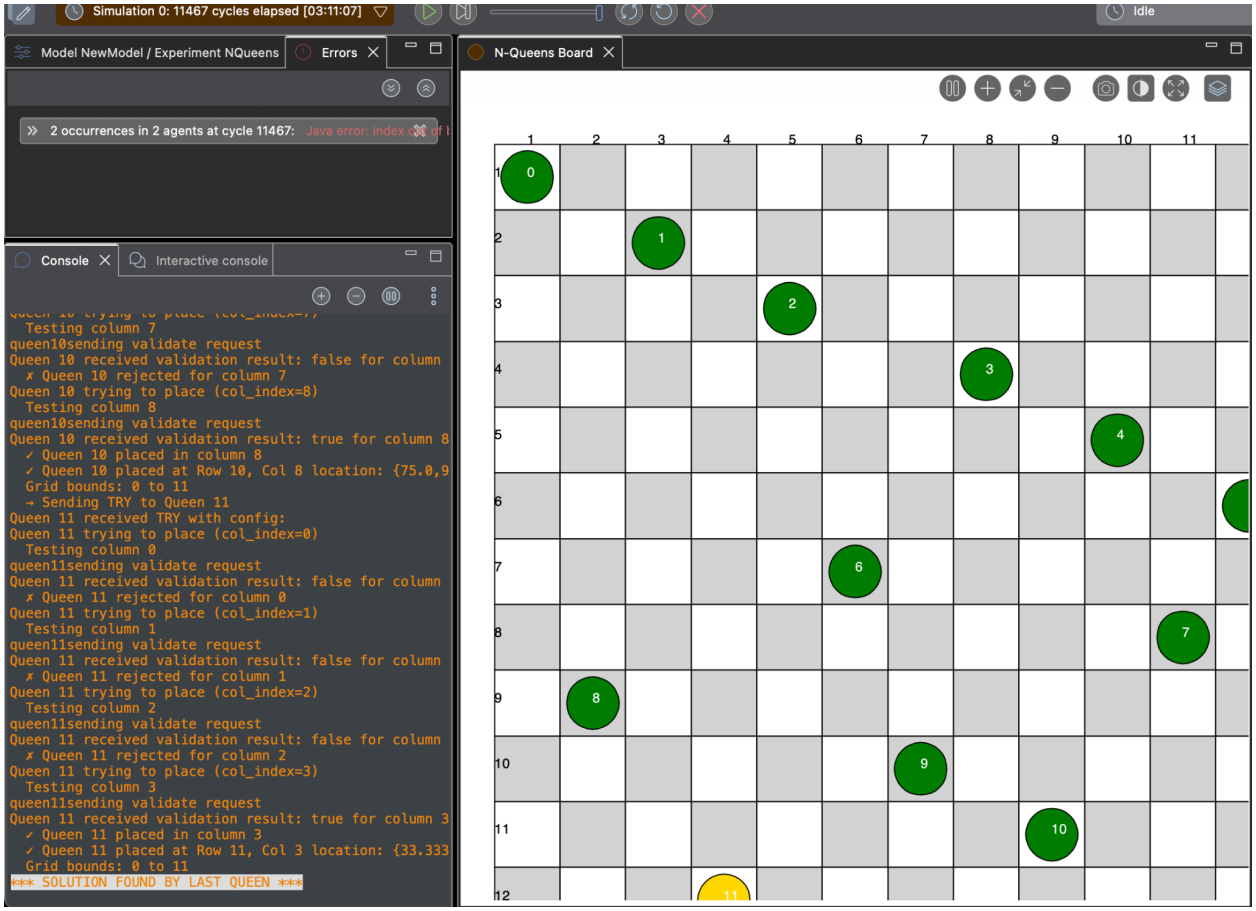




    - **Short interpretation of the result :** A sequence where multiple queens reach conflicting positions and are unable to place themselves successfully. Queen 9 exhausts all available columns and triggers a backtracking request, which forces Queen 8 to retry its placement as well.

This cascading backtracking process illustrates how the agents cooperate to correct earlier decisions and search for a conflict-free arrangement. The board visualization confirms that several queens are marked in red, indicating failed placements, while others remain correctly positioned as the system works toward a valid solution.

- **Use Case - 2 : Successful Solution Found**
    - **Description :** The final queen finds a valid position, the system completes a full conflict-free configuration, marking the discovery of a correct N-Queen solution
    - **Screenshot of program execution/output :**



    - **Short interpretation of the result :** The queens successfully coordinate their placements across the board, with each queen finding a valid column after validation by its predecessors. Minor conflicts occur but are resolved quickly, allowing the sequence to continue smoothly. When the final queen is placed without conflicts, the system confirms that a complete and valid

N-Queen solution has been achieved, as shown by all queens appearing in green positions on the board.

5. Section : Positioning Speakers at Main Stage

5.1 : Explanation:

In this task we focus on extending the festival environment by introducing multiple stages that guests can visit. Each stage offers different attributes, such as light show quality, speaker strength and music style, and each guest has individual preferences for these attributes. The aim of the task is for guests to autonomously choose which stage to attend by evaluating how well each stage matches their personal preferences.

To solve this, we create Stage agents that publish their attribute values and Guest agents that use FIPA communication to request this information. When a guest begins the selection process, it queries all stages and receives their attribute data. The guest then calculates a utility score for each stage by combining the stage's attributes with its own preference weights. Once all utilities are computed, the guest selects the stage with the highest score, moves toward it and spends a set duration there. This process repeats over time, allowing guests to make dynamic, preference-driven decisions while the model keeps track of overall stage visits and popularity.

5.2 : Code:

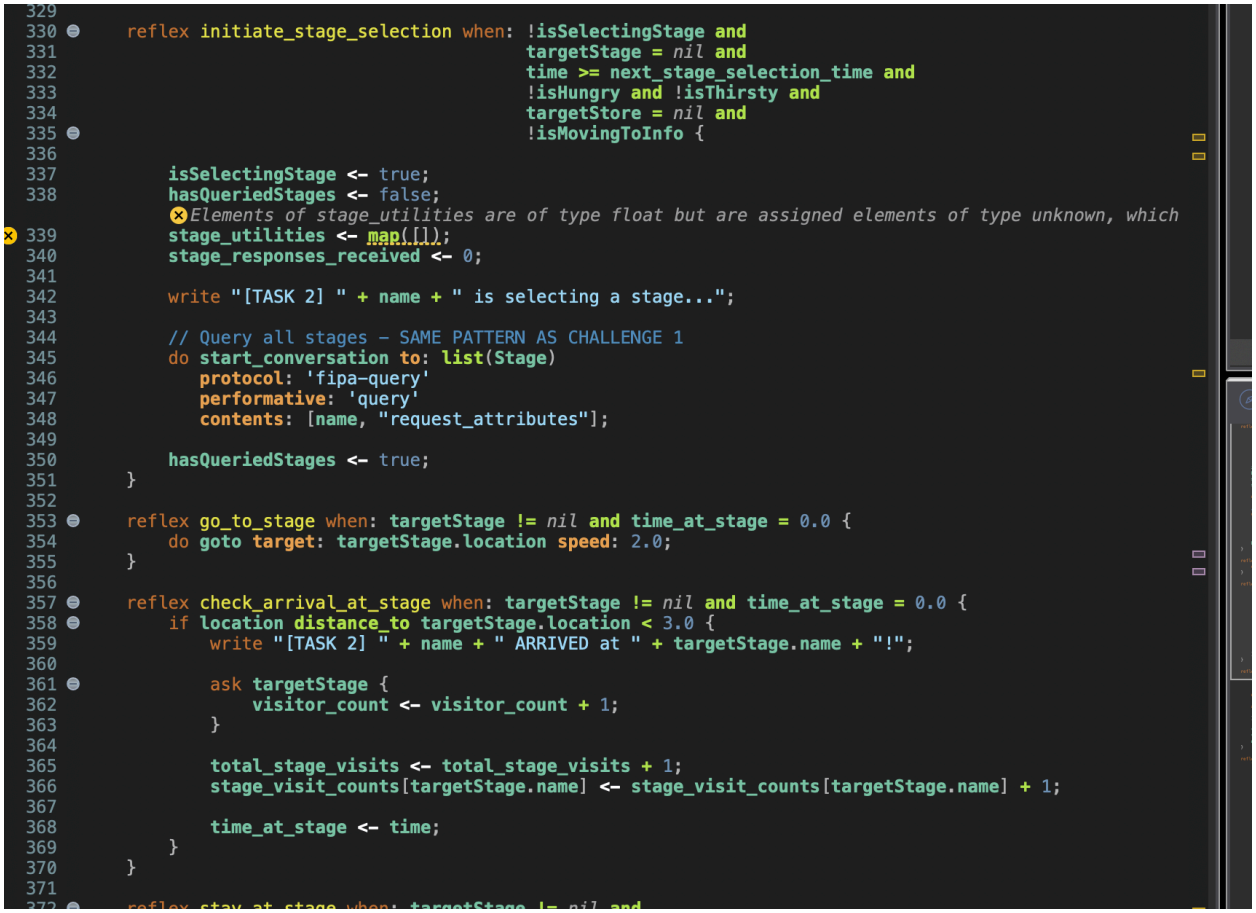
- **Relevant Code Snippet(s) :**

```
// Stage responds with its attributes
species Stage skills: [fipa] {
   float lightShow <- rnd(0.2, 1.0);
   float speaker <- rnd(0.2, 1.0);
   float musicStyle <- rnd(0.2, 1.0);

   reflex respond_to_queries when: !empty(queries) {
      loop q over: queries {
         do start_conversation to: [q.sender]
            protocol: 'fipa-query'
            performative: 'inform'
            contents: ["stage_info", name, lightShow, speaker, musicStyle];
      }
```

```
    }
}

// Guest queries stages, computes utilities and selects best one
species Guest skills: [moving, fipa] {
    float pref_lightShow <- rnd(0.1, 1.0);
    float pref_speaker <- rnd(0.1, 1.0);
    float pref_musicStyle <- rnd(0.1, 1.0);

    Stage targetStage <- nil;
    map<string, float> stage_utilities <- map([]);
    int stage_responses_received <- 0;

    reflex initiate_stage_selection when: targetStage = nil {
        stage_utilities <- map([]);
        stage_responses_received <- 0;

        do start_conversation to: list(Stage)
            protocol: 'fipa-query'
            performative: 'query'
            contents: [name, "request_attributes"];
    }

    reflex receive_inform when: !empty(informs) {
        loop msg over: informs {
            list d <- list(msg.contents);
            if d[0] = "stage_info" {
                string sName <- string(d[1]);
                float u <- pref_lightShow*float(d[2]) +
                           pref_speaker*float(d[3]) +
                           pref_musicStyle*float(d[4]);

                stage_utilities[sName] <- u;
                stage_responses_received <- stage_responses_received + 1;

                if stage_responses_received = length(Stage) {
                    string best <- stage_utilities.keys with_max_of (stage_utilities[each]);
                    targetStage <- Stage first_with (each.name = best);
                }
            }
```

```
    }
  }

  reflex go_to_stage when: targetStage != nil {
    do goto target: targetStage.location speed: 2.0;
  }
}
```

- **Code Screenshots:**



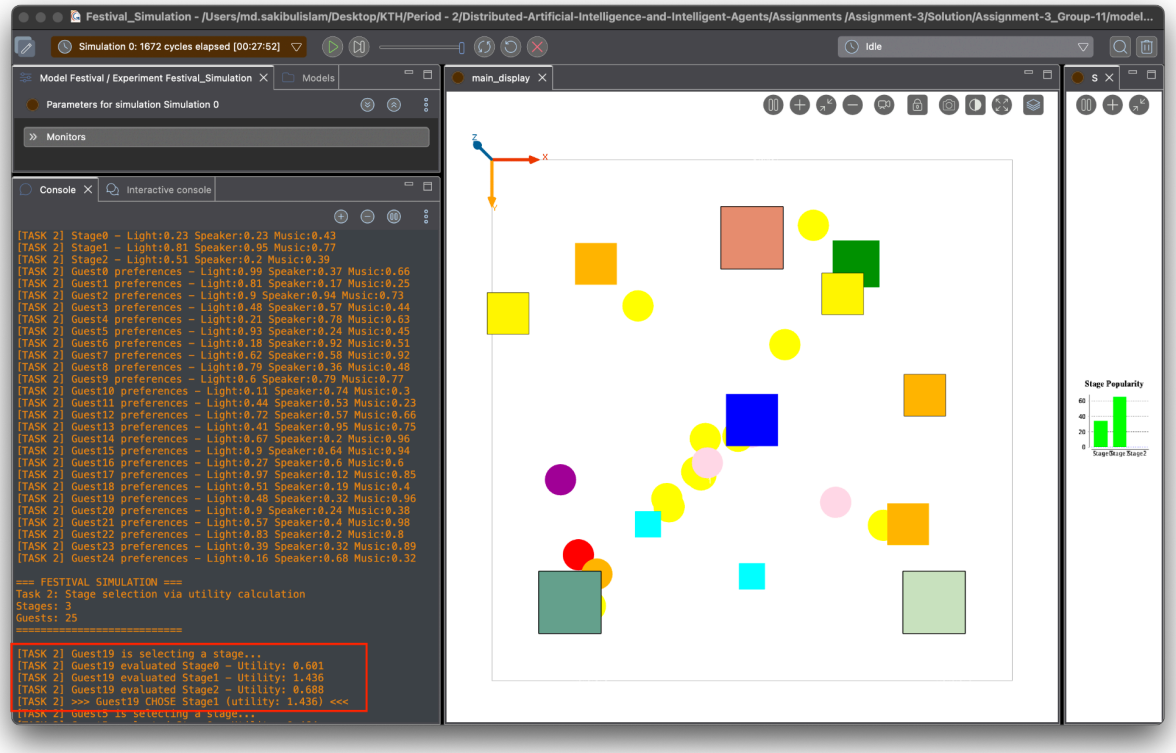
```
    reflex receive_inform when: !empty(informs) {
        loop inform_msg over: informs {
            list data <- list(inform_msg.contents);

            if length(data) >= 1 {
                string msg_type <- string(data[0]);

                // TASK 2: Handle stage information
                if msg_type = "stage_info" and length(data) >= 5 and isSelectingStage and hasQueriedStage
                    string stage_name <- string(data[1]);
                    float s_light <- float(data[2]);
                    float s_speaker <- float(data[3]);
                    float s_music <- float(data[4]);

                    // Calculate utility
                    float utility <- (pref_lightShow * s_light) +
                                     (pref_speaker * s_speaker) +
                                     (pref_musicStyle * s_music);

                    stage_utilities[stage_name] <- utility;
                    stage_responses_received <- stage_responses_received + 1;

                    write "[TASK 2] " + name + " evaluated " + stage_name +
                          " - Utility: " + utility with_precision 3;

                    // Once all stages responded, pick best one
                    if stage_responses_received >= length(Stage) and !empty(stage_utilities) {
                        string best_stage_name <- stage_utilities.keys with_max_of (stage_utilities[each]
                        float best_utility <- stage_utilities[best_stage_name];

                        Stage chosen_stage <- Stage first_with (each.name = best_stage_name);
                        if chosen_stage != nil {
                            targetStage <- chosen_stage;
                            stage_visit_duration <- rnd(50.0, 100.0);
                            time_at_stage <- 0.0;

                            write "[TASK 2] >>> " + name + " CHOSE " + best_stage_name +
                                  " (utility: " + best_utility with_precision 3 + ") <<<";
                        }

                        isSelectingStage <- false;
                        hasQueriedStages <- false;
                    }
                }

                else if msg_type = "winner" and length(data) >= 4 {
                    string auction_id <- string(data[1]);
                    string item_name <- string(data[2]);
```

- **Explanation of the code :** This part of the code handles the entire stage-selection process for each guest. When a guest decides to choose a stage, it resets its selection variables and sends a FIPA query message to all stages asking for their attributes. Once stages reply, the guest collects each response inside receive_inform, extracts the stage's light show, speaker and music values, and calculates a utility score based on its own preferences. Every stage's utility is stored, and when the guest has received all responses, it selects the stage with the highest utility and sets it as the target. The guest then moves to that stage, records its arrival and updates visit statistics. After staying for a set duration, the guest leaves and schedules its next selection time.

5.3 : Demonstration:

- **Use Case - 1 : Guest Selects the Highest-Utility Stage**
    - **Description :** A guest becomes free to choose a stage and sends attribute requests to all available stages. After receiving the replies, the guest calculates utility values based on its own preferences and identifies the stage offering the best match. It then moves directly to that stage, arrives successfully and updates the visit count. This use case demonstrates the normal behaviour of the utility-based selection process, where guests consistently choose the stage that maximises their personal satisfaction.

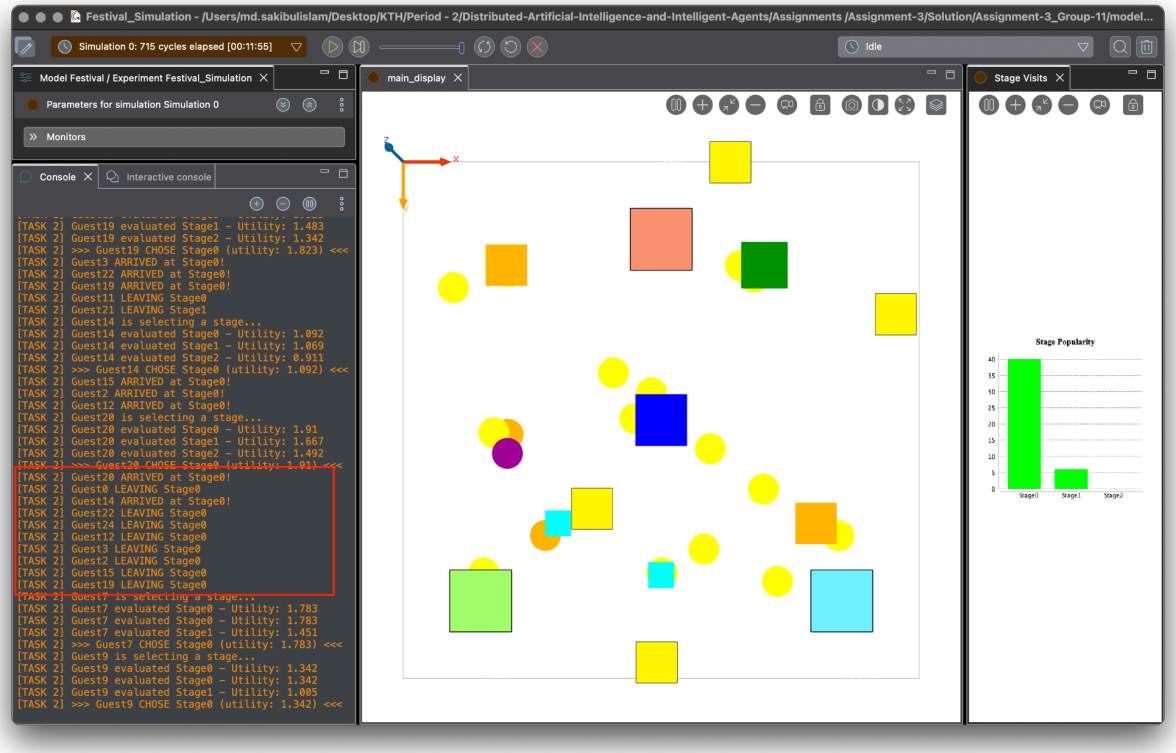
    - **Screenshot of program execution/output :**



    - **Short explanation of the results:** A guest completing the full stage-selection process. The guest queries all available stages, receives their attribute values and calculates a utility score for each one based on its own preferences. After comparing the utilities, the guest chooses the stage that offers the highest match and moves toward it. The highlighted logs confirm each evaluation step and clearly show which stage was selected. This demonstrates that the utility-based selection mechanism is

working as intended, with guests choosing stages that best fit their individual preferences.

- **Use Case - 2 : Multiple Guests Visiting and Leaving the Same Stage**
    - **Description :** Several guests independently select the same stage after calculating their utilities. As they arrive, the stage's visitor count increases, showing how popular stages attract multiple guests at once. Over time, guests finish their visit durations and begin leaving the stage, decreasing the visitor count accordingly. This use case demonstrates the dynamic flow of crowd movement at stages, with guests arriving and departing based on their individual schedules and preferences, while the system accurately tracks and updates stage activity.
    - **Screenshot of program execution/output :**



    - **Short explanation of the results:** Many guests evaluate all stages and consistently choose Stage0 because it gives them the highest utility. Guests arrive, stay for a short duration and then leave, causing the repeated ARRIVED and LEAVING messages. The popularity chart confirms this behaviour, showing Stage0 as the most visited stage.

6. Challenge 1: Global Utility Function

6.1 : Explanation:

This challenge adds a global optimisation problem where guests must consider both their stage preferences and their sensitivity to crowd size. Some guests enjoy large crowds while others prefer quieter spaces, which creates conflicts when everyone chooses independently. A stage that is ideal for one group may become unsuitable for another once it becomes crowded. To address this, guests communicate through FIPA messages and rely on a leader agent who evaluates the overall distribution. The aim is to reach a configuration where the total satisfaction of all guests is as high as possible, even if some individual guests accept a slightly lower personal utility.

Our solution uses a leader-driven optimisation process that improves the global utility step by step. Guests first choose stages on their own, creating an initial distribution. The leader then evaluates whether moving any guest to a different stage would increase the combined utility of all guests, taking crowd preferences into account. If a move improves the global score, the leader instructs the guest to switch and then recalculates the new state. This process repeats until no meaningful improvements remain, at which point the system stabilises and all guests stay at their chosen stages, marking that the global utility has been maximised.

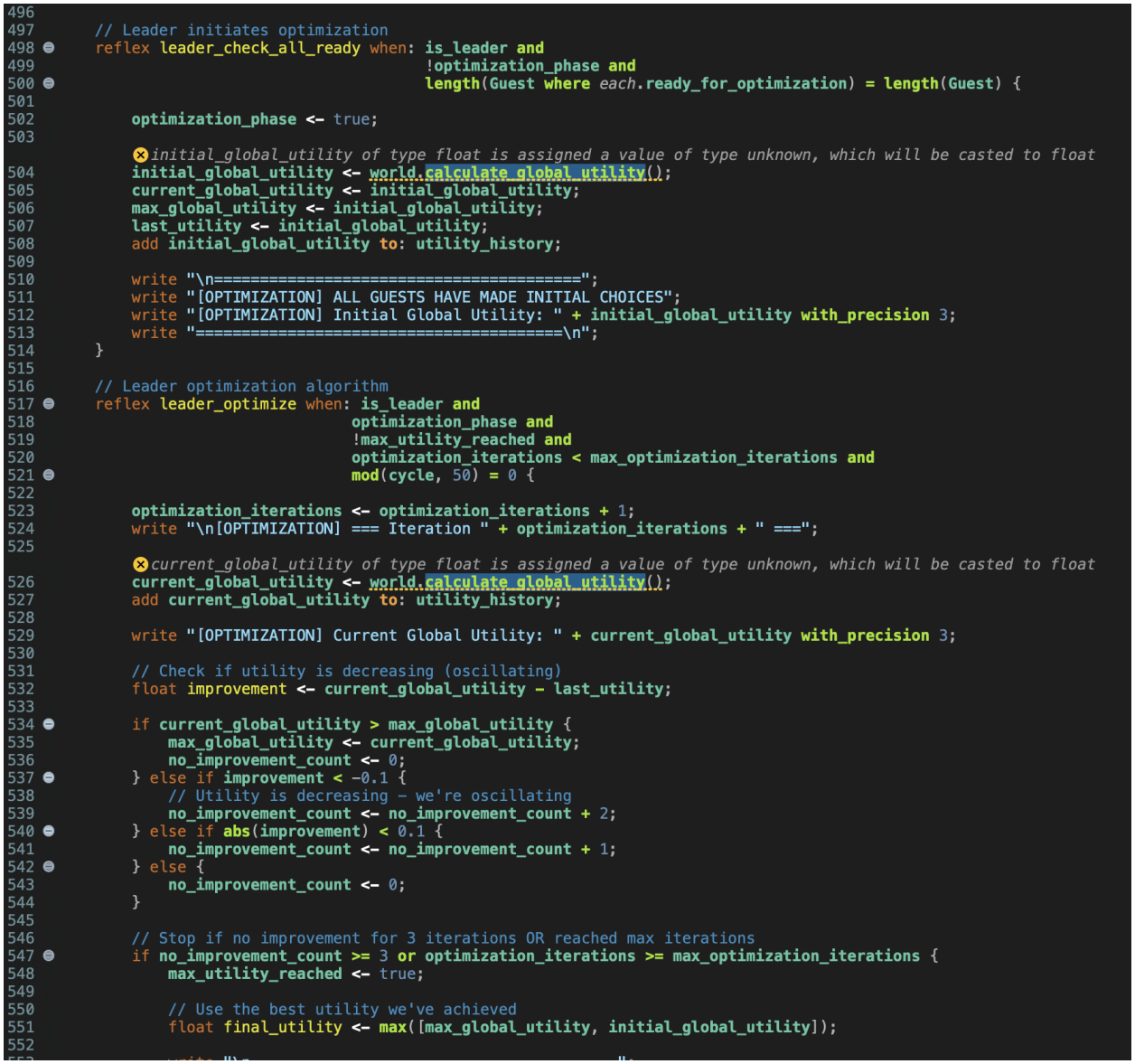
6.2 : Code:

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

```
action calculate_global_utility {
    float total_utility <- 0.0;

    ask Guest {
        if targetStage != nil and time_at_stage > 0.0 {
            float personal_utility <- myself.calculate_guest_utility(self);
            total_utility <- total_utility + personal_utility;
        }
    }

    return total_utility;
}

action calculate_guest_utility(Guest g) {
    if g.targetStage = nil or g.time_at_stage = 0.0 {
        return 0.0;
```

```
        }

        // Base utility from stage attributes
        float base_utility <- (g.pref_lightShow * g.targetStage.lightShow) +
                              (g.pref_speaker * g.targetStage.speaker) +
                              (g.pref_musicStyle * g.targetStage.musicStyle);

        // Crowd mass adjustment
        int crowd_size <- g.targetStage.visitor_count;
        float crowd_factor <- 1.0;

        if g.pref_crowd_mass > 0.7 {
            // Prefers large crowds
            crowd_factor <- 1.0 + (crowd_size / 15.0);
        } else if g.pref_crowd_mass < 0.3 {
            // Prefers small crowds
            crowd_factor <- max(0.3, 1.5 - (crowd_size / 10.0));
        } else {
            // Moderate preference
            crowd_factor <- 1.0 + (0.2 - abs(crowd_size - 8.0) / 20.0);
        }

        return base_utility * crowd_factor;
    }
}
```

- **Code Screenshots:**



```
496
497     // Leader initiates optimization
498     reflex leader_check_all_ready when: is_leader and
499                                         !optimization_phase and
500                                         length(Guest where each.ready_for_optimization) = length(Guest) {
501
502         optimization_phase <- true;
503
            initial_global_utility of type float is assigned a value of type unknown, which will be casted to float
504         initial_global_utility <- world.calculate_global_utility();
505         current_global_utility <- initial_global_utility;
506         max_global_utility <- initial_global_utility;
507         last_utility <- initial_global_utility;
508         add initial_global_utility to: utility_history;
509
510         write "\n=====================================";
511         write "[OPTIMIZATION] ALL GUESTS HAVE MADE INITIAL CHOICES";
512         write "[OPTIMIZATION] Initial Global Utility: " + initial_global_utility with_precision 3;
513         write "=====================================\n";
514     }
515
516     // Leader optimization algorithm
517     reflex leader_optimize when: is_leader and
518                                  optimization_phase and
519                                  !max_utility_reached and
520                                  optimization_iterations < max_optimization_iterations and
521                                  mod(cycle, 50) = 0 {
522
523         optimization_iterations <- optimization_iterations + 1;
524         write "\n[OPTIMIZATION] === Iteration " + optimization_iterations + " ===";
525
            current_global_utility of type float is assigned a value of type unknown, which will be casted to float
526         current_global_utility <- world.calculate_global_utility();
527         add current_global_utility to: utility_history;
528
529         write "[OPTIMIZATION] Current Global Utility: " + current_global_utility with_precision 3;
530
531         // Check if utility is decreasing (oscillating)
532         float improvement <- current_global_utility - last_utility;
533
534         if current_global_utility > max_global_utility {
535             max_global_utility <- current_global_utility;
536             no_improvement_count <- 0;
537         } else if improvement < -0.1 {
538             // Utility is decreasing - we're oscillating
539             no_improvement_count <- no_improvement_count + 2;
540         } else if abs(improvement) < 0.1 {
541             no_improvement_count <- no_improvement_count + 1;
542         } else {
543             no_improvement_count <- 0;
544         }
545
546         // Stop if no improvement for 3 iterations OR reached max iterations
547         if no_improvement_count >= 3 or optimization_iterations >= max_optimization_iterations {
548             max_utility_reached <- true;
549
550             // Use the best utility we've achieved
551             float final_utility <- max([max_global_utility, initial_global_utility]);
552
```

```
        if no_improvement_count >= 3 or optimization_iterations >= max_optimization_iterations {
            max_utility_reached <- true;

            // Use the best utility we've achieved
            float final_utility <- max([max_global_utility, initial_global_utility]);

            write "\n====================================";
            write "[SUCCESS] MAXIMUM GLOBAL UTILITY REACHED!";
            write "[SUCCESS] Initial Utility: " + initial_global_utility with_precision 3;
            write "[SUCCESS] Final Utility: " + final_utility with_precision 3;
            write "[SUCCESS] Improvement: " + (final_utility - initial_global_utility) with_precision 3;
            write "[SUCCESS] Iterations: " + optimization_iterations;

            if no_improvement_count >= 3 {
                write "[SUCCESS] Converged - No more beneficial switches found";
            } else {
                write "[SUCCESS] Maximum iterations reached";
            }

            write "[SUCCESS] All guests can now enjoy their shows!";
            write "====================================\n";

            ask Guest {
                if targetStage != nil and time_at_stage > 0.0 {
                    write "[ENJOY] " + name + " is enjoying the show at " + targetStage.name + "!";
                }
            }

            return;
        }

        last_utility <- current_global_utility;

        // Find beneficial switches that IMPROVE from CURRENT state
        map<Guest, Stage> proposed_switches <- map([]);
        float baseline_global_utility <- current_global_utility;
        float best_found_utility <- baseline_global_utility;

        list<Guest> guests_to_check <- Guest where (each.targetStage != nil and each.time_at_stage > 0.0 and each !=

        loop guest_to_optimize over: guests_to_check {
            Stage current_stage <- guest_to_optimize.targetStage;
            Stage best_stage <- current_stage;
            float best_global_utility <- baseline_global_utility;

            list<Stage> alternative_stages <- list(Stage) where (each != current_stage);

            loop test_stage over: alternative_stages {
                // Simulate switch
                int original_test_count <- test_stage.visitor_count;
                int original_current_count <- current_stage.visitor_count;

                test_stage.visitor_count <- test_stage.visitor_count + 1;
                current_stage.visitor_count <- current_stage.visitor_count - 1;

                Stage temp_original <- guest_to_optimize.targetStage;
                guest_to_optimize.targetStage <- test_stage;

                test_global_utility of type float is assigned a value of type unknown, which will be casted to floa
```

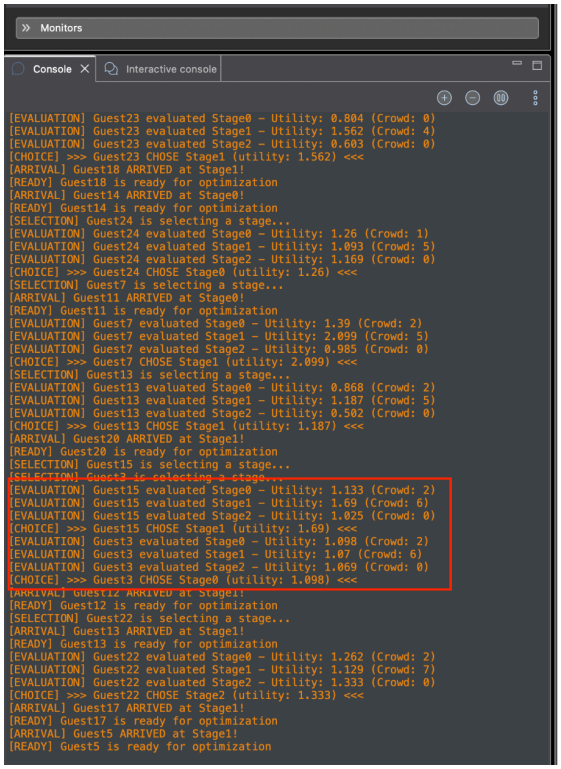
```
                test_global_utility of type float is assigned a value of type unknown, which will be casted to flo
605             float test_global_utility <- world.calculate_global_utility();
606
607             // Restore
608             test_stage.visitor_count <- original_test_count;
609             current_stage.visitor_count <- original_current_count;
610             guest_to_optimize.targetStage <- temp_original;
611
612             // Only accept if it improves SIGNIFICANTLY from baseline
613             if test_global_utility > best_global_utility + 0.5 {
614                 best_global_utility <- test_global_utility;
615                 best_stage <- test_stage;
616             }
617         }
618
619         if best_stage != current_stage and best_global_utility > best_found_utility {
620             proposed_switches[guest_to_optimize] <- best_stage;
621             best_found_utility <- best_global_utility;
622             write "[SWITCH] " + guest_to_optimize.name + " should switch to " + best_stage.name +
623                   " (utility gain: " + (best_global_utility - baseline_global_utility) with_precision 3 + ")";
624         }
625     }
626
627     if !empty(proposed_switches) {
628         write "[OPTIMIZATION] Executing " + length(proposed_switches) + " switches";
629
630         loop g over: proposed_switches.keys {
631             Stage new_stage <- proposed_switches[g];
632
633             ask g {
634                 pending_switch_stage <- new_stage;
635                 received_switch_command <- true;
636             }
637         }
638     } else {
639         write "[OPTIMIZATION] No beneficial switches found";
640         no_improvement_count <- no_improvement_count + 1;
641     }
642 }
643
644 // Execute pending switch
645 reflex execute_switch when: pending_switch_stage != nil and targetStage != nil and time_at_stage > 0.0 {
646     write "[MOVING] " + name + " switching from " + targetStage.name + " to " + pending_switch_stage.name;
647
648     ask targetStage {
649         visitor_count <- visitor_count - 1;
650     }
651
652     targetStage <- pending_switch_stage;
653     time_at_stage <- 0.0;
654     pending_switch_stage <- nil;
655     received_switch_command <- false;
656 }
657
658 reflex update_utility when: targetStage != nil and time_at_stage > 0.0 {
        my_current_utility of type float is assigned a value of type unknown, which will be casted to float
659     my_current_utility <- world.calculate_guest_utility(self);
660 }
661
662 reflex trackDistance {
```

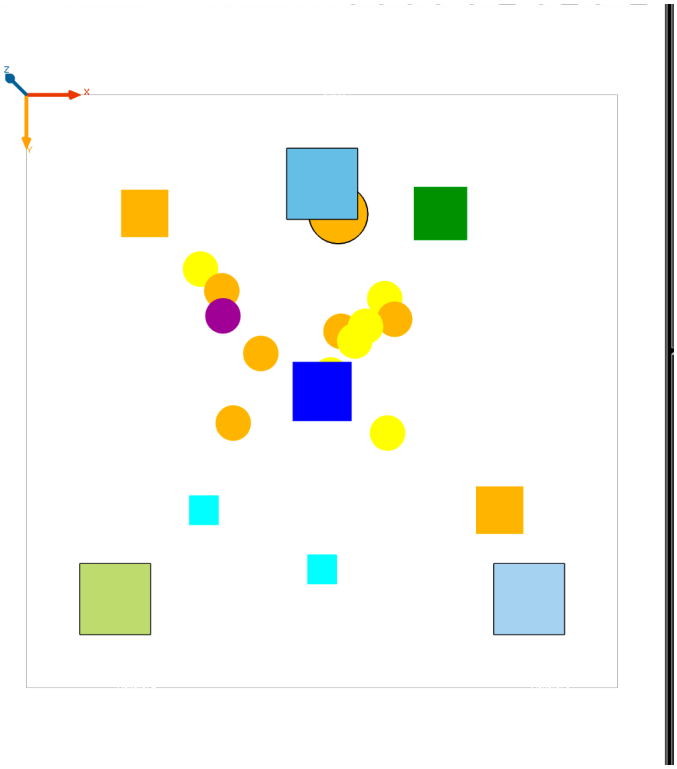
- **Explanation of the code :** It handles the leader's global utility optimisation process. When all guests are ready, the leader enters optimisation mode, calculates the initial global utility and begins iterating. In each iteration, the leader recomputes the current global utility, checks if it is improving and stops if no progress is made after several rounds. The core logic tests every guest by temporarily moving them to alternative stages and recalculating the global utility to see whether the switch would benefit the whole system. Only moves that significantly increase the global utility are kept. If beneficial switches are found, the leader sends commands to the corresponding guests to relocate. The process repeats until no more helpful moves exist or the maximum number of iterations is reached. When the optimisation stabilises, the leader announces success and guests remain at their final stages.

6.3 : Demonstration:

- **Use Case - 1 : Guests Choose Stages Based on Both Utility and Crowd Size**
    - **Description :** A guest evaluates all stages by combining the stage attributes with the current crowd size. If the crowd level aligns with the guest's crowd preference, the utility increases. Guests who enjoy larger crowds tend to choose busier stages, while those who prefer quieter environments select stages with fewer people.

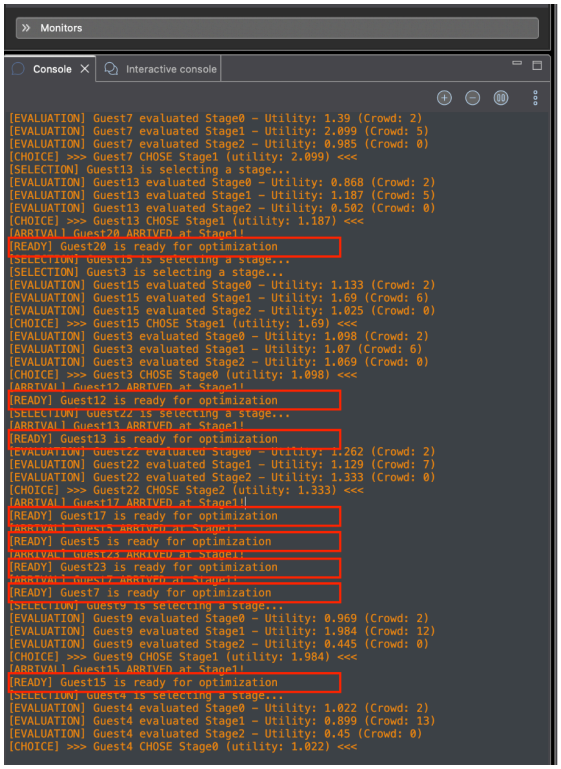
    - **Screenshot of program execution/output :**

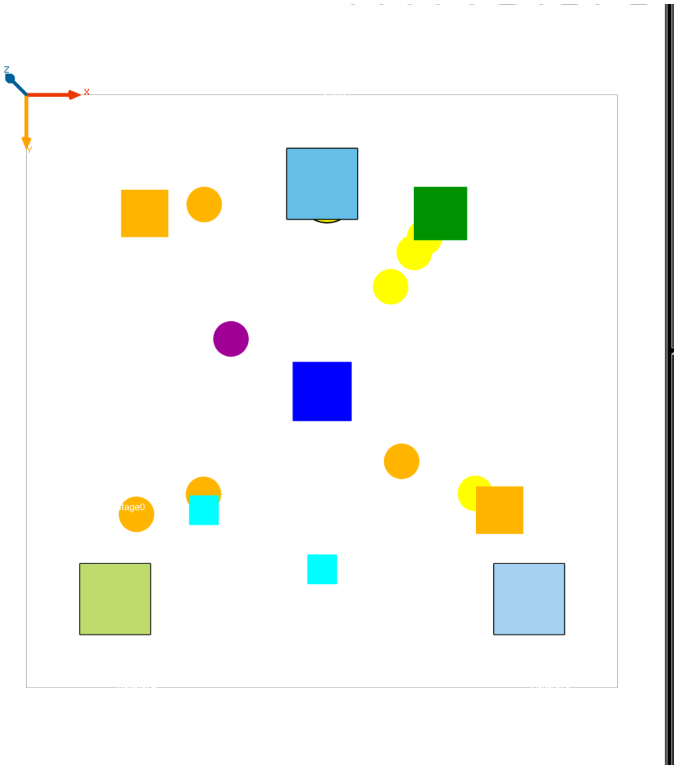




    - **Short explanation of the results:** Here guests evaluate all stages by considering both stage attributes and current crowd sizes. Each guest selects the stage that gives the highest overall utility based on its individual crowd preference. Guests who enjoy larger groups choose the busier stage, while those who prefer quieter settings select stages with fewer people. This demonstrates that the crowd-mass preference is correctly influencing stage selection, leading different guests to choose different stages even when evaluating the same options.

- **Use Case - 2 : Guest Signals Readiness for Optimization**
    - **Description :** After completing its stage selection and arriving at a chosen stage, a guest marks itself as ready for the optimisation phase. This allows the leader to track who has settled into a stage and ensures that optimisation only begins once every guest has reached this state.
    - **Screenshot of program execution/output :**



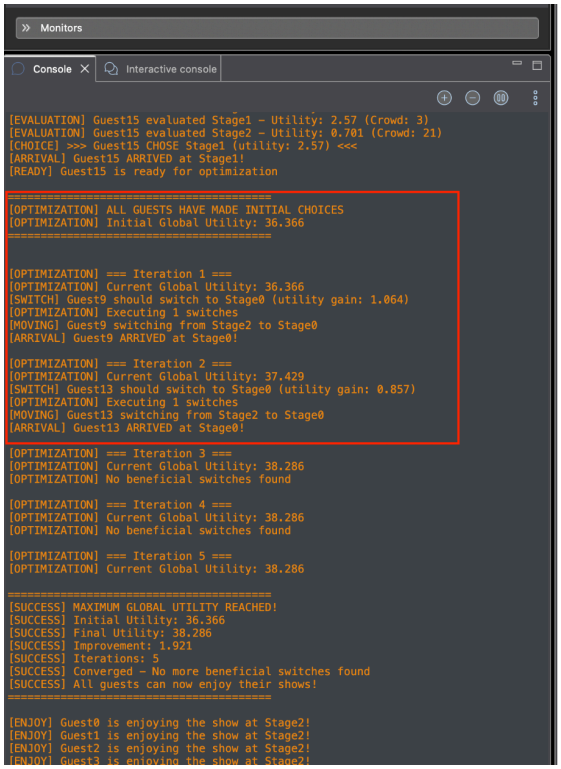


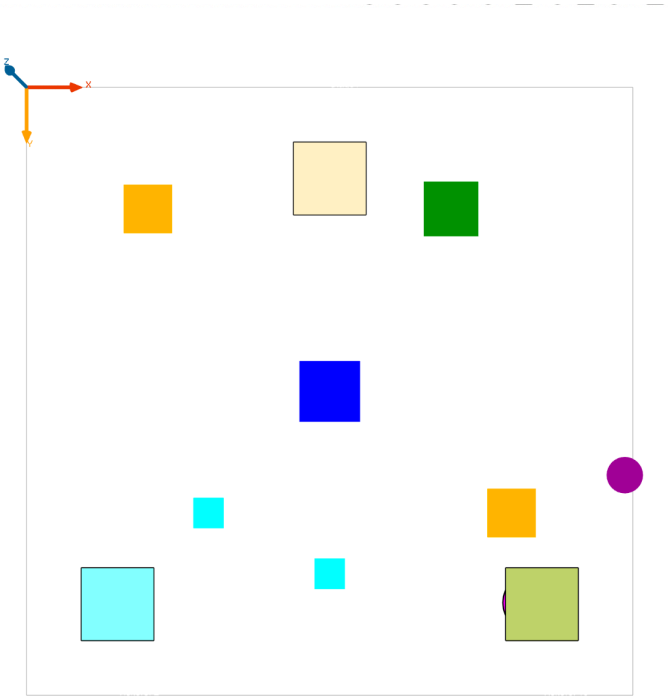
    - **Short explanation of the results:** It shows several guests completing their stage selection and announcing that they are ready for the optimisation phase. Each guest first evaluates all stages, chooses the one with the highest utility and arrives there. Once settled, they report themselves as ready. This indicates that the system has reached the point where every guest is positioned at a stage and the leader can now begin the global optimisation process.

- **Use Case - 3 : Guests Moves to Increase Global Utility**
    - **Description :** During the optimisation iterations, the leader detects that one guest can improve the global utility by switching to another stage. The leader issues the switch command, and the guest moves from its current stage to the suggested one. After arriving, the system updates the utility

values, showing that the switch produced a positive gain for the overall group.

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



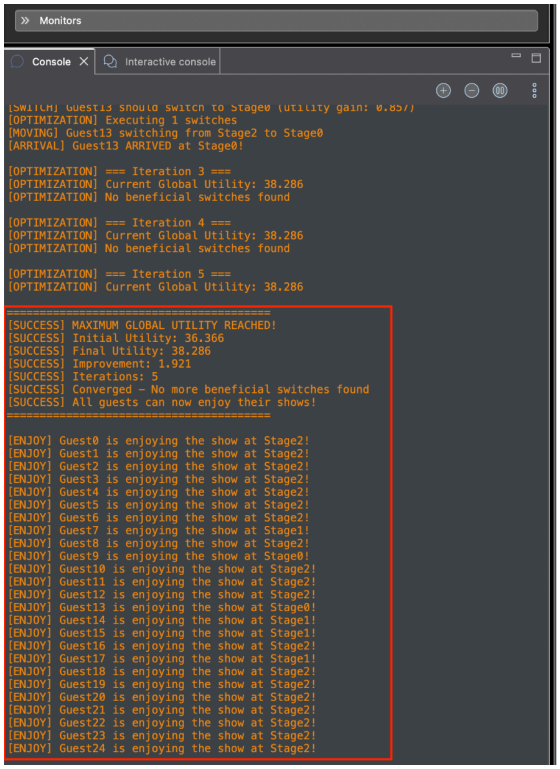


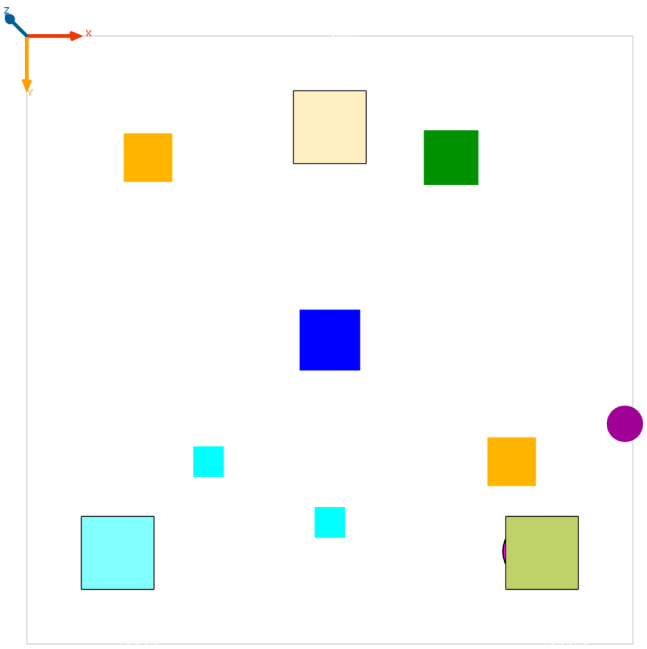
- **Short explanation of the results:** The optimisation phase begins after all guests have made their initial stage choices. The leader calculates the global utility and then identifies guests whose relocation would increase overall satisfaction. In the first two iterations, beneficial switches are found, and the leader instructs those guests to move to a better stage. Each switch raises the global utility until no further improvements are possible. The process then stops, confirming that maximum global utility has been reached and all guests stay at their final stages.

- **Use Case - 4 : Guests Settle Into Their Final Optimised Stages**

  - **Description :** Once the optimisation completes, each guest stays at the stage that best fits the globally optimised configuration. Guests no longer move or recalculate utilities and simply enjoy the show at their assigned stages. This demonstrates the stable end state of the system after achieving maximum global utility.

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





- **Short explanation of the results:** The optimisation process has fully completed and the system confirms that no further improvements are possible. The final global utility is higher than the initial one, showing that the switches made during optimisation were beneficial. All guests have now settled at the stages that contribute to the maximum global utility, and each guest remains at its final stage enjoying the show.

7. Final Remarks :

- **Summarize :**
  - In Task 1 we built a fully agent-based solution to the N Queens problem, where each queen acted independently and communicated only with its neighbouring agents. Through distributed validation and backtracking, the queens cooperatively found conflict free board configurations without any central controller.
  - Task 2 expanded the festival model by introducing stages with different attributes and guests who selected stages based on utility calculations. Guests used FIPA communication to gather information and made choices that reflected their individual preferences, resulting in a dynamic and believable simulation.

- Challenge 1 added a global optimisation layer by introducing crowd preferences and requiring guests to maximise collective satisfaction rather than only individual utility. A leader agent coordinated this process by analysing possible guest relocations and guiding the system toward the configuration with the highest total utility. Together these tasks demonstrate how autonomous agents can cooperate, communicate and optimise both local and global objectives within a complex multi agent environment.

- **Limitations or Improvements :**
  - For Task 1 (N-Queen), one limitation was the complexity of handling message-based backtracking. Because every queen depends on predecessor validation, even a small logical mistake in message passing could break the entire search. We also noticed that when many queens were stuck, the console output became difficult to follow. An improvement would be to add clearer visual debugging or reduce repeated log messages. Another improvement would be performance tuning for larger N values, since the message chains grow long. A smaller design limitation was the reliance on strict sequential communication; in future, parallel consistency checks could reduce waiting time.
  - For Task 2 (Stage Selection), the main challenge came from combining stage selection with other guest behaviours like hunger, thirst, and auctions. Several reflexes conflicted with each other, causing guests to get stuck or ignore stage selection. The improvements we made included fixing FIPA protocol handling, adjusting reflex conditions, cleaning console output, and improving the priority of movement behaviours. We also simplified message handling so stage information and auction messages no longer interfere with each other. Another improvement added was visual feedback to show whether guests were selecting, travelling, visiting stages, or participating in auctions, which made debugging much easier.
  - For Challenge 1 (Global Utility Optimization), the main limitation was the complexity of coordinating all guests through a leader. We faced multiple issues including wrong protocol handling, oscillating switches, leader death errors, guests not moving after optimization commands, and overlapping responsibilities inside the receive_inform reflex. Step by step, these were resolved with specific improvements: consolidating message handling into a single reflex, adding oscillation detection, raising the improvement threshold, slowing optimization cycles, ensuring pending switch execution works reliably, re-electing a new leader when needed, and cleaning message queues to prevent backlog. These improvements

allowed the leader to guide the system toward convergence without infinite loops or incorrect switches.

- **Comments :** Overall, the assignment offered a good mix of agent communication, coordination and optimisation. Each task built on the previous one and helped us understand how individual behaviours interact within a larger multi agent system. The debugging process also played an important role, since many issues only appeared once multiple systems were running together. Despite the complexity, the final system runs smoothly and demonstrates the intended behaviours clearly.