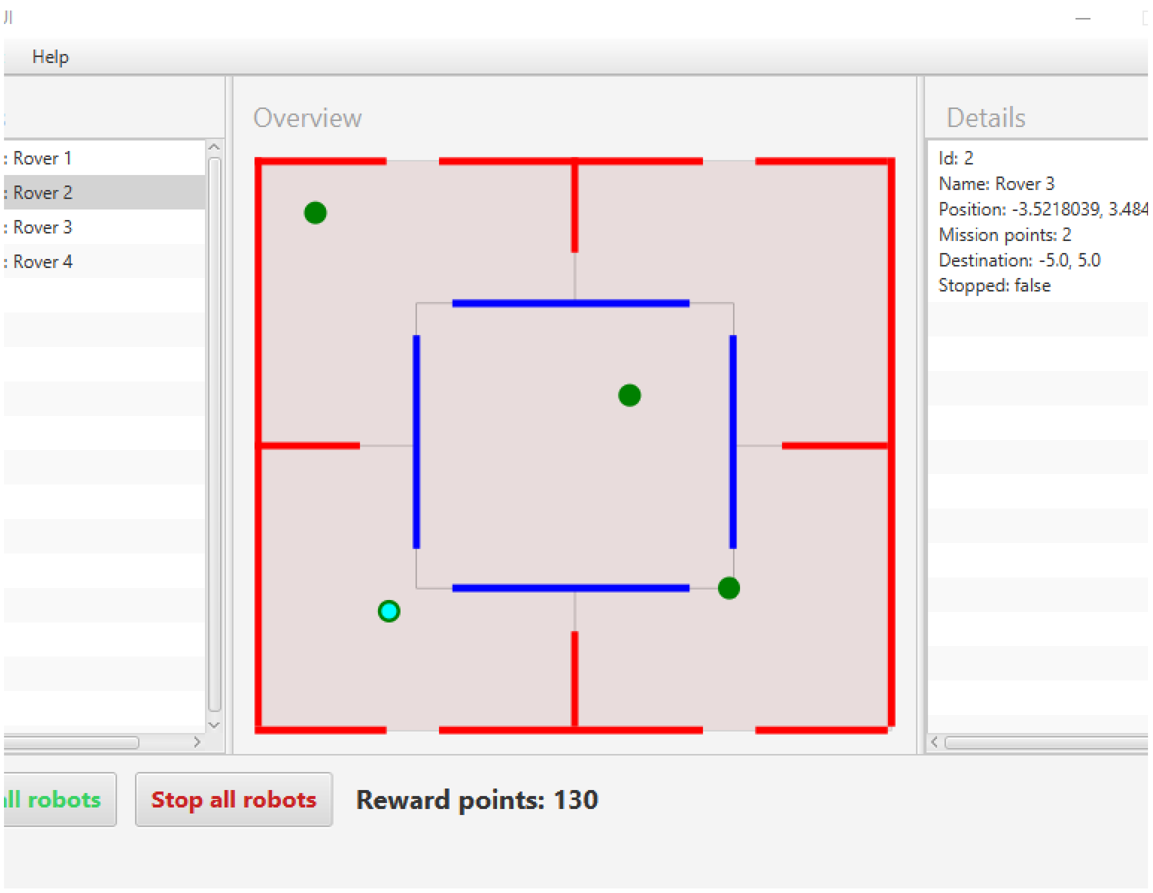


Id: 2
Name: Rover 3
Position: -3.5218039, 3.484
Mission points: 2
Destination: -5.0, 5.0
Stopped: false

Reward points: 130

Final Report

Model-Driven Software Development

Group 16 | DIT945/TDA593 | 2019-01-09
Rasmus Tomasson rastom@student.chalmers.se,
Rikard Teodorsson rikardt@student.chalmers.se,
Mattias Torstensson mattors@student.chalmers.se,
Rema Salman remas@student.chalmers.se,
Pontus Eriksson ponerik@student.chalmers.se

Table of Contents

Introduction

For this project modeling techniques are used to model a robotic system, where Section 1 describes the system's context, domain model, and use cases. In Section 2 the software architecture of the project is described by decomposing the ROVU system into a component diagram, see figure 1. It consists of components and their relationships as interfaces and operations, which will be described by the following headings. Section 3 contains further decisions of the system implementation represented in a class diagram. Section 4 describes the feature model, which is used by the clients to customize the system according to their needs. Section 5 contains sequence diagrams that describe the main three behaviors of the system.

Definitions, Acronyms, and Abbreviations

UML	Unified Modeling Language
CS	Central Station
ID	Identification
SPL	Software Product Line
APP	Application
UI	User Interface
GUI	Graphical User Interface
CLI	Command Line Interface

1. System Overview

In this section the system's context, domain model, and use cases are defined, where UML-based modelling techniques were used for the domain model that conceptualized the robotic system ROVU. Moreover, the use case diagram and its scenarios' are described.

1.1 System Context

The ROVU system contains rovers, i.e. robots that autonomously move within a designated environment, e.g. a hospital. An interface allows an operator to monitor the environmental status that rovers perform their missions in, and display the positions of the rovers and reward points. The missions are performed sequentially by rovers according to the hospital environment, where they simultaneously start moving with an initial placement of one in each surgery room, to then one at a time enter the consulting room. Upon entering the room, each rover stops for two seconds and then returns to the surgery room. Additionally, each mission is composed of strategies that follow sequential points, which are based on a graph representation of the environment for allowing the rovers to use a pathfinding algorithm. Furthermore, the reward system computes the rewards every 20 seconds regardless of the environment. The points given to rovers, the system's environment, and the system's areas, are pre-assigned. Thus, to calculate the reward points all the rovers are checked if they are inside the areas; if so the points are added according to those areas. Each rover is responsible to continuously report its position to the central station, and check if there are any obstacles in its way.

1.2 Domain Model

Figure 1 shows the domain model, where proximity sensors, wheels, a camera, and a GPS sensor are parts of a rover. The model was created by relating the extracted concepts, which existed in the provided project description, to each other. The proximity sensors enable the rover to detect obstacles in the environment, for example other rovers and/or boundaries. Every rover has a network device for communicating with the central station. Further, the central station has a common GUI that displays the combined gained reward points by the rovers. The central station handles procedures by switching between them, calculates the rewards, and updates the rovers' heading if they have the RewardMaximizer behavior. Several rovers move in an environment that contains obstacles. Each environment includes various areas, each area being either logical or physical and may overlap other areas. The physical areas are bounded by walls while the logical areas can stretch across boundaries. The missions performed by one rover are to be completed in that environment. Missions are executed based on one or several strategies according to the order of points that are used by the strategy/strategies. Rovers can perform missions in various and different environments, however, the team decided to only have one environment at a time when using the system in our domain model.

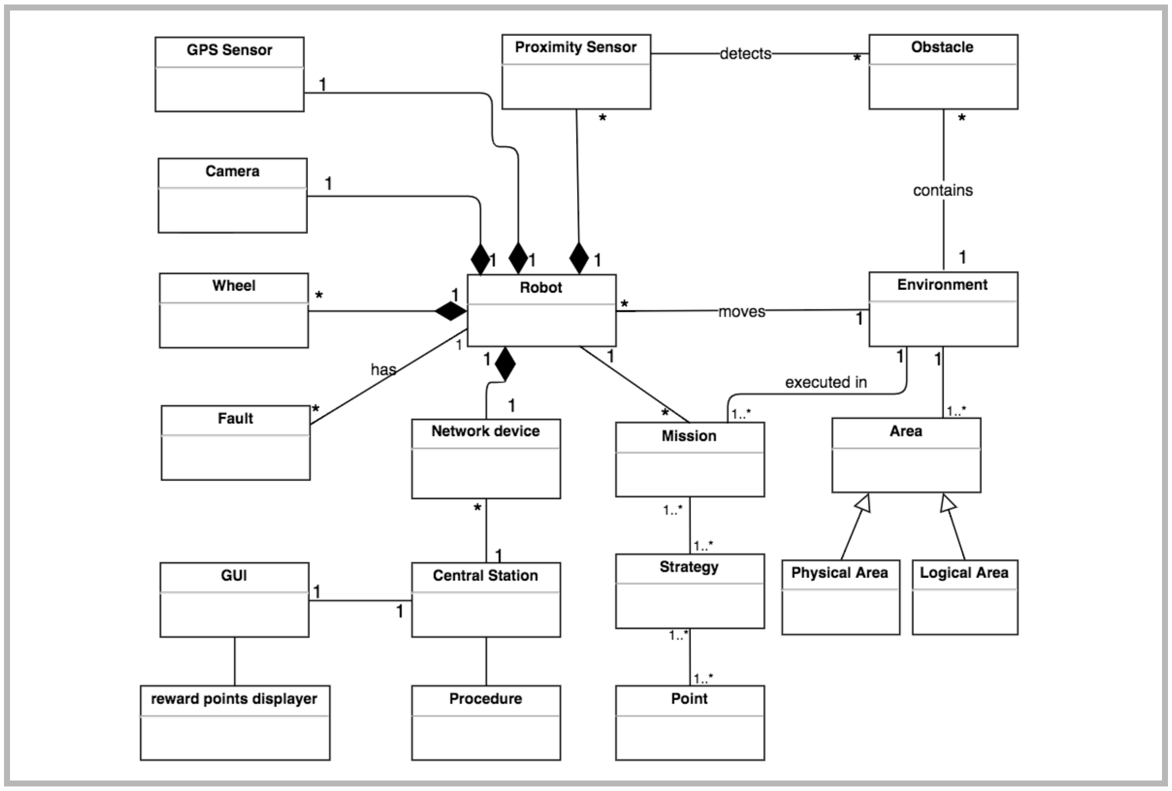


Figure 1. Domain model of the robotic system.

1.3 Functional Requirements

Figure 2 depicts the use case diagram of the ROVU system, and how the different actors Operator, Rover, and Central Station, interact with the system for executing a desired result of the system. Each use case represents a specific functional requirement of the system [1]. In the case of combining and executing all the use cases, the basic flow of the system will be achieved. Simultaneously, the figure below shows the extension points between the main use cases: include and extend, in order to simplify the figure and make it more understandable. However, a further description for each use case is given in the next sub-section.

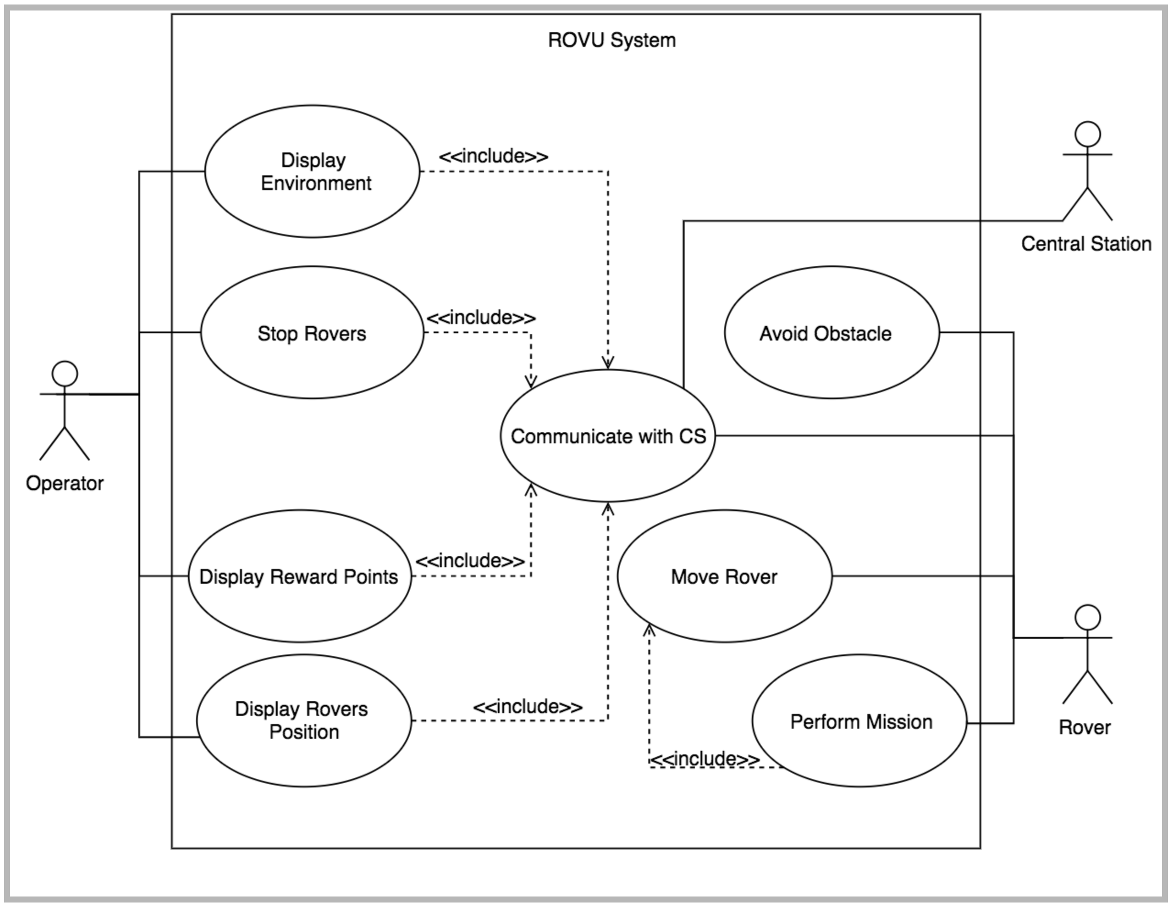


Figure 2. Use case diagram of the ROVU system.

1.4 Use Case Actors

The main actors are the users of the system (operators), the central station that all communications are made through, and the rover. The right side of figure 2 has the actors that are part of the system, but in this case, are considered as secondary or supporting actors for the system's representation. On the other hand, the operator is the primary actor of the system.

1.5 Basic Flow

The basic flow of the system is a result of combining all use cases displayed in figure 2. The operator starts the system, then correspondingly the rovers will simultaneously start executing their movement and perform their missions within the environment. The operator is able to stop the rovers, as well as see the reward points along with the rovers' positions in the environment. Moreover, the reward points are calculated according to the areas each rover is located in, after which they are combined and displayed on the user interface. Each rover communicates its position with the central station to prevent other rovers from entering the same room at the same time. Sequentially, the central station includes all the communications that are held between rovers.

1.6 Use Case Model

The use cases previously shown in figure 2 have their corresponding goal, actor, and description (basic flow) provided separately below.

➢ Display Environment

Actor: An Operator.
Goal: See the environment in which the robots are located.
Description: The operator is able to see an environment that is displayed through the interface.
Use Case Inclusion: Communicate with Central, due to the fact that the interface (GUI) depends on the central station for displaying the system's environment.

➢ Display Reward Points

Actor: An Operator.
Goal: See the reward points displayed on the interface.
Description: An operator uses the interface to see the combined reward points gained by the robots.
Use Case Inclusion: Communicate with Central, due to the fact that the interface (GUI) depends on the central station for displaying the combined reward points.

➢ Display Rover Position

Actor: An Operator.
Goal: See the environment displayed on the interface.
Description: An operator uses the interface to see the location of the robots while they execute their missions.
Use Case Inclusion: Communicate with Central, due to the fact that the interface (GUI) depends on the central station for displaying the rovers' positions and movements.

➢ Stop Rovers

Actor: An Operator.
Goal: Uses the interface to stop the system.
Description: An operator immediately stops (pauses) the rovers' missions by pressing the

stop button in the interface.
Use Case Inclusion: Communicate with Central, due to the fact that the interface (GUI) depends on the central station for stopping the rovers.

➢ Communicate with CS

Actor: Robot.
Goal: Each rover communicates with the central to perform the simultaneous movements and missions execution.
Description: The central station communicates with the rovers by providing their missions and their executions. Furthermore, the station is responsible for demanding the score calculator to start performing as well as getting the combined reward points, so it can be displayed on the interface.

➢ Move Rover

Actor: Robot.
Goal: The robot is capable to move from one position to another.
Description: The robot executes movements according to the mission its completing.

➢ Avoid Obstacle

Actor: Robot.
Goal: The robot avoids obstacles when performing a mission.
Description: The robot is capable of avoiding obstacles, such as other robots and physical environmental boundaries.

➢ Perform Mission

Actor: Robot.
Goal: The robot executes missions according to the environment it exists in.
Description: The robot executes missions according to several points defined as a strategy, where the environment is considered as well.
Use Case Inclusion: Move Rover, due to the fact that the mission performing is dependent on the rover movement.

2. System Structure

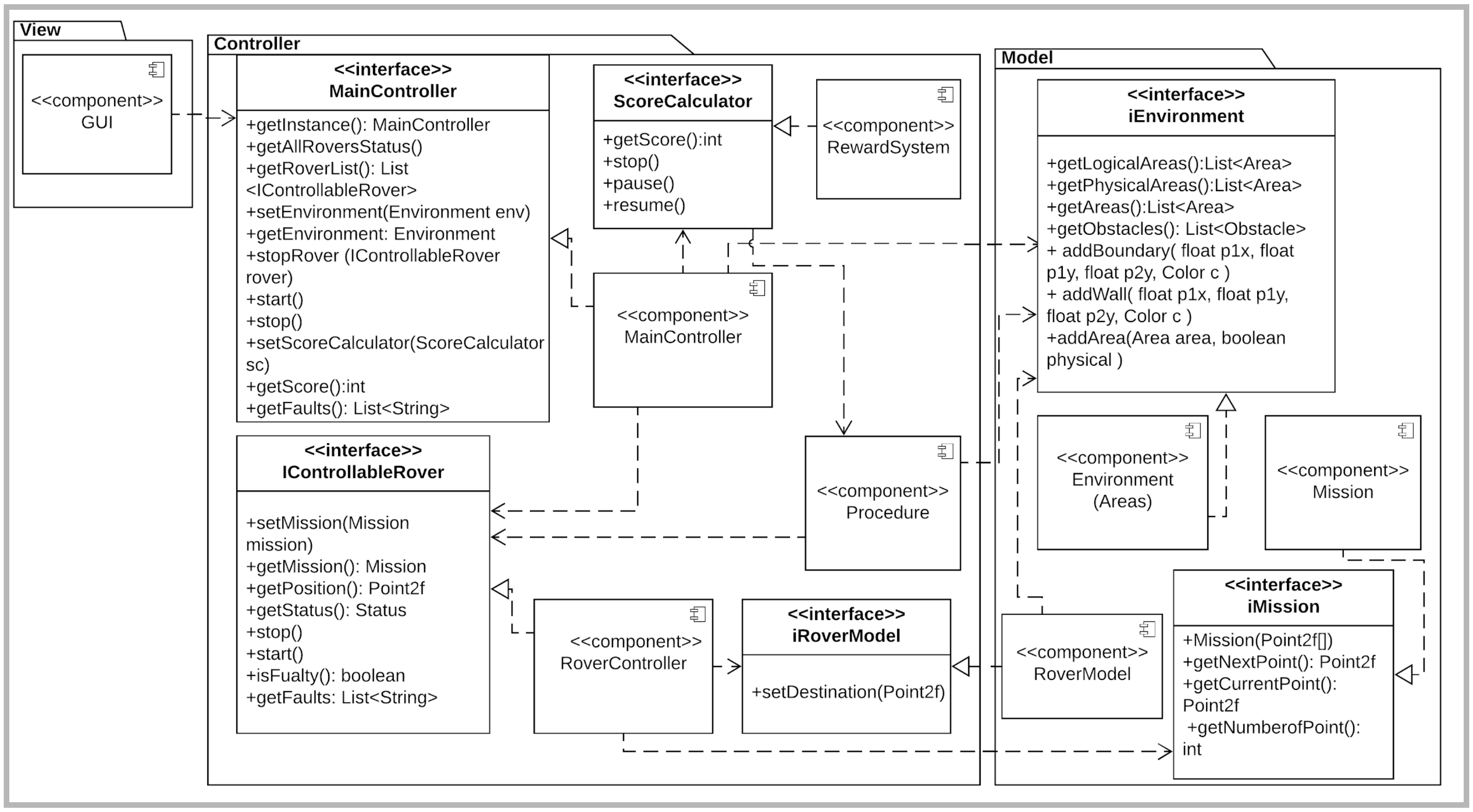
This section focuses on the system's structural/logical view; it clarifies the architectural decision which is briefly explained in the corresponding sub-section. Additionally, an analysis of each component from a high-level view and the components' relationships with each other through the operations encapsulated within the interfaces.

2.1 Architectural Style Decision

Figure 3 shows the determined architectural style Model-View-Controller (MVC) that was chosen according to the system's functionality. Firstly, the View layer displays all the operations executed by the system in the user interface, so the component within it is visible to the operator, which enables him or her to interact with the system. Secondly, due to the rovers' autonomous movement and missions assignment in the system, the Controller layer contains all the logical operations executed by the system and this layer acts as a liaison between the View and Model layers. Ultimately, the Model layer reflects all the data needed to create essential components in ROVU system, e.g. environments, rovers, missions, and scores.

2.2 Component Diagram

The software architecture of the project is described by decomposing the ROVU system into a component diagram, see figure 3. It consists of the system's components and their interactions/relationships as interfaces and operations, which will be described by the following sub-sections.



View
<<component>> GUI
Controller
<<interface>> MainController
+getInstance(): MainController
+getAllRoversStatus()
+getRoverList(): List <IControllableRover>
+setEnvironment(Environment env)
+getEnvironment: Environment
+stopRover (IControllableRover rover)
+start()
+stop()
+setScoreCalculator(ScoreCalculator sc)
+getScore():int
+getFaults(): List<String>
<<interface>> ScoreCalculator
+getScore():int
+stop()
+pause()
+resume()
<<component>> RewardSystem
<<component>> MainController
<<interface>> IControllableRover
+setMission(Mission mission)
+getMission(): Mission
+getPosition(): Point2f
+getStatus(): Status
+stop()
+start()
+isFualty(): boolean
+getFaults: List<String>
<<component>> Procedure
<<component>> RoverController
<<interface>> iRoverModel
+setDestination(Point2f)
Model
<<interface>> iEnvironment
+getLogicalAreas():List<Area>
+getPhysicalAreas():List<Area>
+getAreas():List<Area>
+getObstacles(): List<Obstacle>
+ addBoundary(float p1x, float p1y, float p2y, Color c)
+ addWall(float p1x, float p1y, float p2y, Color c)
+addArea(Area area, boolean physical)
<<component>> Environment (Areas)
<<component>> Mission
<<interface>> iMission
+Mission(Point2f[])
+getNextPoint(): Point2f
+getCurrentPoint(): Point2f
+getNumberofPoint(): int
<<component>> RoverModel

Figure 3. Component diagram of the ROVU system.

2.3 Description of Components

- **GUI**

  This component is the operator's way of both viewing the current status of the system, as well as controlling it, e.g. sending a “stop all rovers”-signal, or updating the environment of the rovers.

- **MainController**
  The main core controller for the ROVU system. It serves to handle all the calculations regarding the other components, such as calculating the score based on the active procedure, as well as controlling the information coming from the interfaces described below.

- **RoverController**
  This component represents a model describing each rover with its controller with the variables, e.g. current position, and speed. It also provides the logic for the rovers functionalities, e.g. pathfinding and obstacle avoidance functions for driving.

- **RewardSystem(Score)**
  The component represents the score in the system, which is used by the reward points displayer.

- **Procedure**
  This component illustrates the procedure logic, used to calculate the reward points of the rovers accordingly to the environment they are in.

- **Environment**
  This component imitates the environment model, i.e. the areas and the borders that are incorporated into the environment’s creation.

- **Mission**
  The component represents the mission model in the system, which is assigned to the rovers by their controller.

- **RoverModel**
  It represents the rovers in the system, with their functionality to drive in the environment.

2.4 Description of Interfaces and Operations

- **MainController**
  The MainController interface enables the GUI to get information about the system that can be handled in the MainController component.
  - getInstance(): MainController
    Returns an instance of the component to be used in another one as an inside controller.
  - getAllRoversStatus()
    This is for getting the rovers’ status.
  - setEnvironment(Environment)
    sets the currently used environment.

- getEnvironment(): environment
  Gets the currently used environment.
- getRoverList(): List<IControllableRover)
  Returns a list of all the rovers in the system.
- stopRover(IControllableRover rover)
  Stops that current rover execution until further instructions are given.
- setScoreCalculator(ScoreCalculator sc)
  Sets the MainController's score calculator. This can be changed at runtime to allow more flexibility to the scores that are added according to the environment/areas.
- stop(): void
  For stopping all rovers and pausing the reward calculator.
- start(): void
  For starting all rovers' execution and resuming the reward calculator.
- getScore(): int
  Returns the current score, and if nothing has been added it returns 0.

iRoverModel

This interface allows the communication between the RoverController and the RoverModel components. The interface implements the Rover model that is responsible for the rover's movement by operating the setDestination(Point2f), which allows the rover to drive to the point that is passed as an argument.

iControllableRover

This interface implements the iControllableRover component, which is used for controlling the rovers and their functional executions. It uses the rover model, while Procedure and MainController use it; it operates the following:

- setMission(Mission mission)
  Sets a mission to the rover, so it starts executing.
- getMission(): Mission
  Returns the current mission.
- getPosition(): Point2f
  Returns the rover's current position in the current environment.
- getStatus(): Status
  Returns the current state of the rover, such as its current objective, if it has been stopped or not, if it is suffering from any faults, name etc.
- stop(): void
  Stops the rover from taking any action until start() is called.
- start(): void
  Starts the rover if it is stopped.
- isFualty(): boolean
  It returns a boolean if a fault has been detected in the rover.

- getFaults(): List<String>
  Returns a list of strings consisting of all the current faults in the rover.

- ## ScoreCalculator
  The ScoreCalculator interface enables the MainController to execute various operations for calculating the score.
  - getScore(): int
    Returns the current score to MainController to be displayed.
  - pause(): void
    It pauses the procedure of the score calculation until a resuming is asked.
  - resume(): void
    continues the procedure of the score calculation where it was left.
  - stop(): void
    Stops the procedure of the score calculation.

- ## iEnvironment
  The iEnvironment interface is used by the MainController component for controlling the rovers as well as by the RoverModel to deal with the positioning of the rovers.
  - getLogicalAreas(): List<Area>
    Returns a list of all the logical areas.
  - getPhysicalAreas(): List<Area>
    Returns a list of all the physical areas.
  - getAreas(): List<Area>
    Returns a list of all the areas.
  - getObstacles(): List<Obstacle>
    Returns a list of all the obstacle.
  - addBoundary(float p1x, float p1y, float p2y, Color c)
    For creating both horizontal and vertical boundaries to the environment.
  - addWall(float p1x, float p1y, float p2y, Color c)
    For creating both horizontal and vertical walls to the environment.
  - addArea(Area area, boolean physical)
    For adding the areas as physical or logical to the environment.

- ## iMission

  This interface is used by the RoverController, and it has the following operations:

  - Mission(Point2f [])
    Creates a new mission from an array of points, where the passed argument represents the points that need to be visited.
  - getNextPoint(): Point2f
    Returns the next point that needs to be visited by the rover according to the mission.

- getCurrentPoint(): Point2f
  Returns the current point that has just been visited.
- getNumberofPoints(): int
  Returns the total number of points in the mission.

3. System development

The section contains more descriptions and explanations of each software component used in the source code according to the implementation, which supports the project's maintainability and extensibility. Hence, this section presents a low-level abstraction of the component diagram of the **ROVU** system, see figure 3, where the abstraction is illustrated by a class diagram, the design patterns followed and transformed into implementation. In addition to that, the deviations between the class model and code.

3.1 Class Diagram

This section illustrates a lower abstraction of design decisions for the system's implementations represented in the class diagram, see figure 4. The diagram aligns with the provided code implementations and structure that is done by the team and can be found on the team's GitHub repository in the "mdsd" folder. https://github.com/remasalm/MDD-group16.

The arrows in the diagram visualize the UML dependency, realization and generalization relationships [1] indicated between the implemented classes. The highest used is the dependency, where the class with the arrow's end is used in the other pointed to as an attribute. The inheritance create the realizations between the interfaces and the classes that implement them. The generalization can be seen on the diagram, such as the Hospital class is specializing and extending the Environment class, where the Hospital inherits attributes, operations and associations.

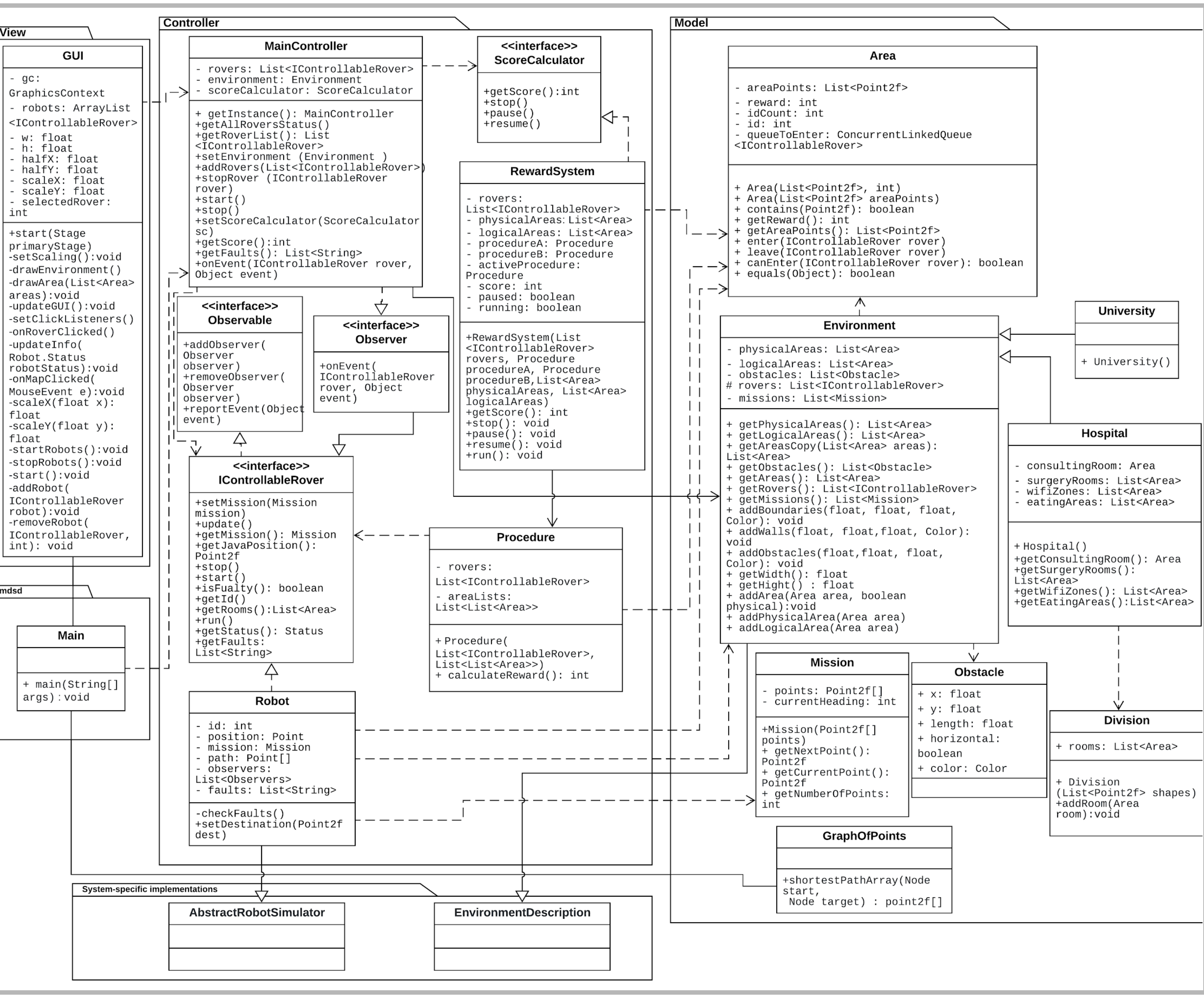


Figure 4. Class diagram of the ROVU system. Link to diagram: lucidchart.com

3.2 Design Patterns

- Creation patterns

A builder Design pattern is represented by the separated complex objects such as the environment into areas, division, and obstacles.

- **Structural patterns**

  A bridge pattern is produced by the published interfaces in the inheritance hierarchy to allow the implementation to be used among multiple objects. The bridge pattern also allows for decoupling between the core system and the implementation of the simulator's system by providing an abstraction layer between the two systems. For example, this can be found by the connections between the content of "System-specific implementations" package and both "Robot" and "Environment" classes.

- **Behavioral patterns**

  An observer pattern is used where the robots automatically notify the observer (IControllableRover) of any faults in the robots.

  A strategy pattern is used to make it possible to have multiple easily interchangeable protocols used in the system.

3.3 Deviations between Model and Code

- The "System-specific implementations" package is a lot more intricate than displayed in the diagram but is presented in a simplified version for showing the external existence of Simbad's Library/plug-in and its connection with the system's classes.

3.4 Test cases

JUnit 5 was used for writing the test classes where the code consists of four classes and a test suite to run all of them. They can be found in the src/test/ folder. The test cases are separately explained in the following subheadings;

- **MissionTest**

  In this class, the test cases mainly test the current point, the next point, and the number of points for a mission.

- **PathfindingTest**

  This test checks that the path returned after running the pathfinding algorithm on a predefined graph is in fact the shortest path for travelling between two of the points in that graph.

- **ProcedureTest**

  This test class contains two cases for calculating the rewards. The first checks the area and according to it, the reward is added. The second case tests the initialization of the rewards' points when the rovers are not in areas or before moving.

- **RewardSystemTest**

  It consists of test cases for checking the functionalities in its corresponding class, such as getting the score, pause, resume, and stop. Further, several cases when running the reward system, for example: Case 1 tests the changed procedure from A to B in logical Areas, Case 2 tests the changed procedure from B to A in logical Areas, Case 3 tests the changed procedure from A to B in physical Areas, and Case 4 tests the change procedure from B to A in physical Areas.

- **RobotTest**

  This class tests a robot's functionality in terms of assigning missions, getting the mission, start, stop, update, and the check for if two robots are the same object. Additionally, several cases for mission's execution by a rover, where it checks all areas if a rover has entered to a new room, the rover leaves that area, where it gets removed from the queue, and if the rover does not enter any new room.

4. Software Product Line (SPL)

This section contains the Software Product Line (SPL) in terms of "features" visualized and described according to the feature mode, see figure 5. It is a representation of all the current features and possible future products as extensions. A justification of the feature scoping and the feature implementations are covered in this section as well.

4.1 Feature model

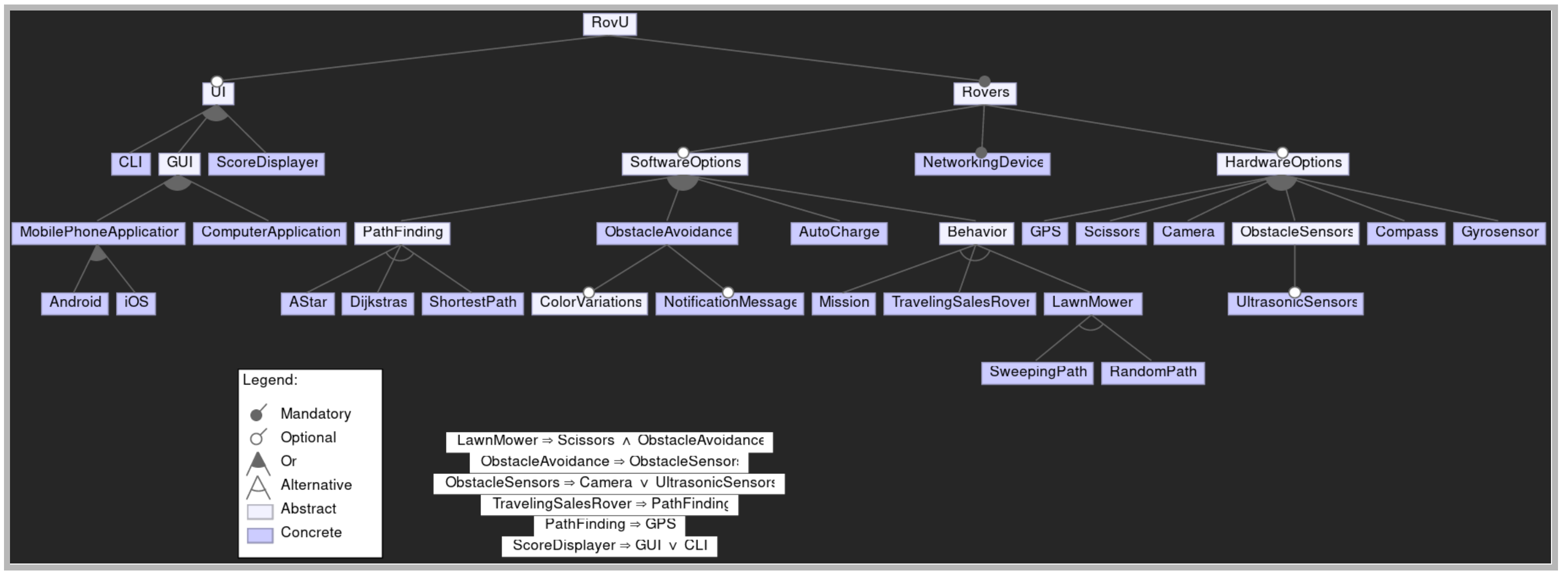


Figure 5. Feature model of ROVU system. Link to the .xml file: drive.google.com

4.2 Description of Features

The following descriptions are indented according to the hierarchy of the features tree, as shown in figure 5.

4.2.1 ROVU

The ROVU system represents the most abstract of the features. Moreover, the customer is required to buy a Rover but can optionally choose to buy a UI, as well as software and hardware options.

4.2.2 UI

The user interfaces for the rover can be either a graphical interface or a command line interface.

- **CLI**

  Command Line Interface - For controlling the system from the command line. Targeted for the more advanced customers with a technical experience. Not as profitable as the GUI but less costly to implement.

- **GUI**

  Graphical User Interface - Control the system with a clean, functional, graphical interface. Targeted specifically at the less technical customer. Likely, most profitable seeing as most customers probably lack technical expertise.

  - Desktop app

    A computer application for the graphical user interface.

  - Mobile phone app

    A mobile phone application for the graphical user interface. A mobile phone is required by the customer.

    - Android

      Android version of the mobile phone app.

    - iOS

      iOS version of the mobile phone app.

- **Score displayer**

  This is used for displaying the score accumulated by the rovers. Thus, it is displayed either on the GUI or in a CLI.

4.2.3 Rovers

This represents the abstract of the rovers, where the customer is required to buy the networking device attached to the rover. However, both the software and the hardware are optional.

- **Software options**

  - AutoCharge

    This feature allows the rover to automatically find some kind of charging station when its battery is low.

  - Pathfinding

    Calculates a path from the rover's current position to an arbitrary position in the environment. Since the rover would need to know where in the environment it is for pathfinding to function, a GPS becomes a requirement. This feature provides several options for various algorithms.

        - Dijkstra’s algorithm

          Finding the shortest paths from the source vertex (edges) to all other vertices (nodes) in a graph [2]. Therefore, it can be used to find the shortest path between two given nodes.

        - A* algorithm

          It can be seen as an extension of Dijkstra’s algorithm, but it achieves better performance [3].

        - Shortest path

          Finding the shortest path between two nodes through the connected edges [4].

    - Obstacle avoidance

      A system that allows the rover to dynamically avoid sudden obstacles that are detected by the rover’s sensors, outside of the pathfinding system. Obstacle avoidance requires some way to detect obstacles, such as a camera or an ultrasonic sensor. It increases the rover’s lifespan by not crashing into obstacles.

        - Color variations

          As an extension for the obstacle avoidance feature, the obstacles have been visualized with various colors in the GUI of the system, to make it easier for the user to see.

        - Notification message

          This is another extension of the obstacle avoidance feature, where the operator gets a notification message as soon as a rover has encountered an obstacle.

    - Behavior

      This is an abstract feature that provides the ability of behavior variation. Everything in this feature group is mutually exclusive with each other since the rover can only have one behavior that it follows.

        - Mission

          For a rover to execute a mission, it should move according to a list of ordered points. This requires one of the algorithms from the pathfinding feature so that the rover can perform the movements.

        - Traveling sales rover

          A solver for visiting a list of points in the fastest order. Requires pathfinding so that the rover can travel to the different points.

        - Lawnmower

Lawnmower behavior for the rover is optimized for lawn mowing. It requires the scissors as additional hardware so that the rover can cut the grass. Furthermore, it also requires obstacle avoidance software so that the rover can change direction upon reaching an obstacle.

- Sweeping Path

  While performing the lawnmower behavior, the rovers systematically cover the entire environment by going back and forth in a sweeping manner.

- Random Path

  This lawn mower behavior means that the rovers drive until they hit an obstacle, then turn to a random new direction, and start driving again.

➢ Hardware options

- GPS
  A GPS, utilized by for example the pathfinding algorithm. This is required in order for the pathfinding system to function.

- Scissors
  Scissors are used for cutting grass. Targeted for the lawn mower behavior.

- Ultrasonic sensors
  Used for detecting the immediate physical surrounding of a rover, which is grouped under the Obstacle Sensor.

- Camera
  Used for seeing the physical surroundings of the rover, or for taking pictures. It can also be used to detect obstacles that obstruct the rover's path.

- Compass
  Used for knowing which direction the rover is currently facing. Useful for the lawn mower which rotates to change direction.

- Gyro sensor
  Used for measuring the rover’s angle relative to the ground, or its velocity. Useful for when the surface the rover is traveling on is not flat.

➢ Networking device

Used for communicating with a remote station via a network, which is a requirement for all rovers.

4.3 Justification of feature scoping

➢ **Pathfinding**

Necessary for finding a path to an arbitrary coordinate-described point, which is a useful feature for a variety of different scenarios, such as a lawnmower style robot returning to a charging station or an idling position.

➢ **Voice Control**

Implementation of a system that allows a rover to be successfully controlled by voice is on its own a project whose scope is larger than that of the current rover project and as such has been excluded due to not being feasible.

4.4 Implementation of Features

- The **Score displayer** is displayed on the GUI for the user.
- The GUI is provided as a **Desktop APP** since it has been developed in JavaFX.
- The sub-feature **Color variations** is provided in the source code, where the obstacles are visualized with a different color in the environment displayed on the GUI.
- The **A* algorithm** was followed to implement the pathfinding feature. It is used to calculate a path, where the environment is predefined as a graph that contains several nodes, and edges connecting between nodes.
- The various points of the **behavior feature** have been implemented in the source code.

5. System behavior

5.1 Sequence diagrams

This section contains the sequence diagrams of the main three behaviors of the system, which exist in the use cases explained earlier in Section 1 “Use Cases”. They are “Change of Strategy”, “Perform Mission”, and “Reward Calculation”.

➢ **Change of Strategy**

The sequence diagram for the change of strategy, figure 6, represents only one rover’s behavior in the system, although it contains multiple rovers. Each rover checks its position within the corresponding Environment. If it has changed its location or entered a different area (areaChanged == true), it stops for two seconds and then starts again. The returned area from getArea is compared with the rover’s current area and stored in areaChanged.

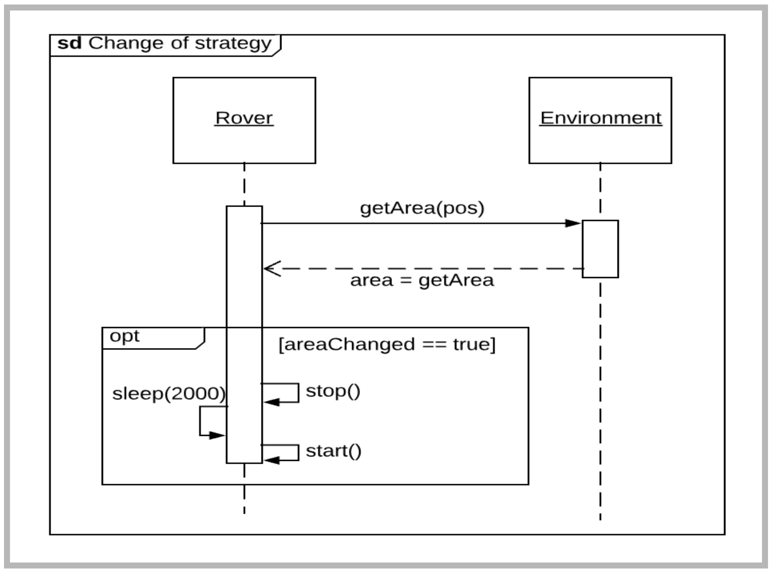


Figure 6. Sequence Diagram for Change of Strategy.

➢ Perform Mission

The sequence diagram for the perform mission, figure 7, represents a rover's behavior in the system. The rover's controller sets the missions and demands the execution, where each of the rovers has a loop. While performing a mission, each rover continuously checks its position and if the current point from the strategy being executed is reached. If so the destination is set to the next point of the strategy.

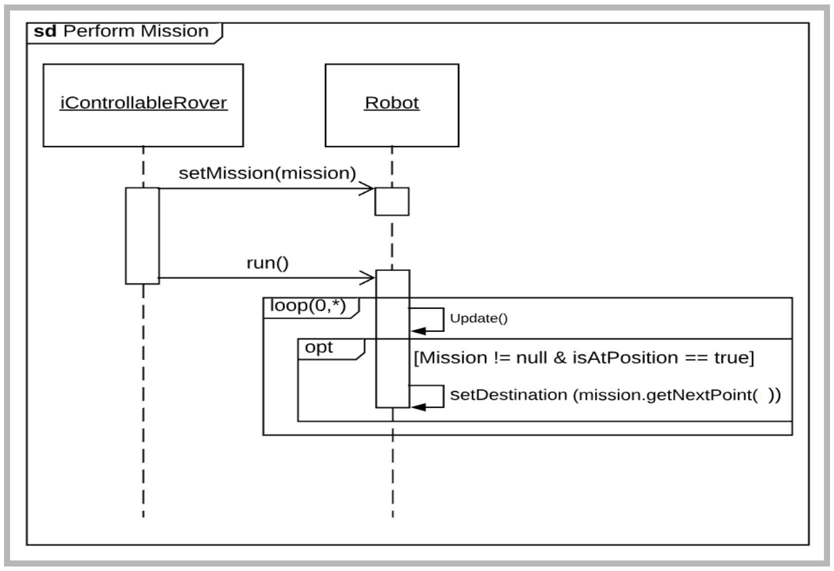


Figure 7. Sequence Diagram of Perform Mission Use Case.

➢ Reward Calculation

Upon starting the score calculator it begins calculating the score using the currently active procedure. After the calculation is done it checks if the criterion for changing the procedure is fulfilled, and if it is, it changes the procedure. Finally, the score calculator sleeps for 20 seconds before starting over again from the start, see figure 8.

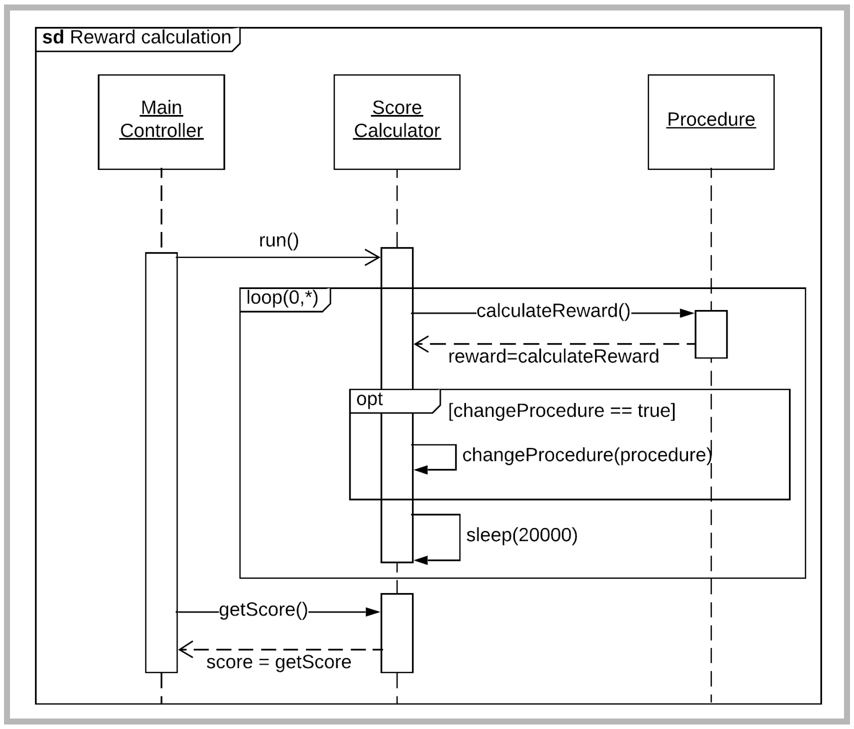


Figure 8. Sequence Diagram of Reward calculation.

References

[1] Larman, C. (2012). *Applying UML and patterns: an introduction to object oriented analysis and design and iterative development*. Pearson Education India.

[2] Wikipedia contributors. (2018, December 18). Dijkstra's algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 19:31, December 20, 2018, from https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=874271298

[3] Wikipedia contributors. (2018, December 20). A* search algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 19:27, December 20, 2018, from https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=874620581

[4] Wikipedia contributors. (2018, November 17). Shortest path problem. In Wikipedia, The Free Encyclopedia. Retrieved 19:35, December 20, 2018, from https://en.wikipedia.org/w/index.php?title=Shortest_path_problem&oldid=869305113