Software design
Team project – Deliverable 3

Team number: 28

Team members:

Name	Student Nr.	E-mail
Fabiola Loffredi	2617929	f.loffredi@student.vu.nl
Carmen Toyas Sánchez	2622278	c.toyassanchez@student.vu.nl
Roshini Ramasamy	2610864	r.ramasamy@student.vu.nl
Cristiano Milanese	2600169	c.milanese@student.vu.nl
Anna Claire Favié	2626585	a.c.favie@student.vu.nl

This document has a maximum length of 15 pages.

Table of Contents

1. The new requirement

Author(s): Fabiola Loffredi

For the new requirement, we were asked to modify our robot so that it can react differently to different types of hardware fault. For this task, we did not have to do any particular modification to the code itself, rather we only added the new edited requirements.

When doing so, we had to take into account that hardware faults are completely random, and may not be very frequent. Therefore, we relied on a random number generation, as it is not possible to manually "break" the robot with some built-in command. Moreover, to make the faults as "rare" as possible, we decided to assign hardware faults only to numbers with the same digits.

The program keeps generating a random number between 100 and 999999, whose digits are compared using a simple algorithm which counts the number of times the same digit appears. If the generated number has all the same digits, then a different fault is triggered, according to the length of the number (which is stored in `counter`). In our case, we generated three possible outcomes and consequent behavior:

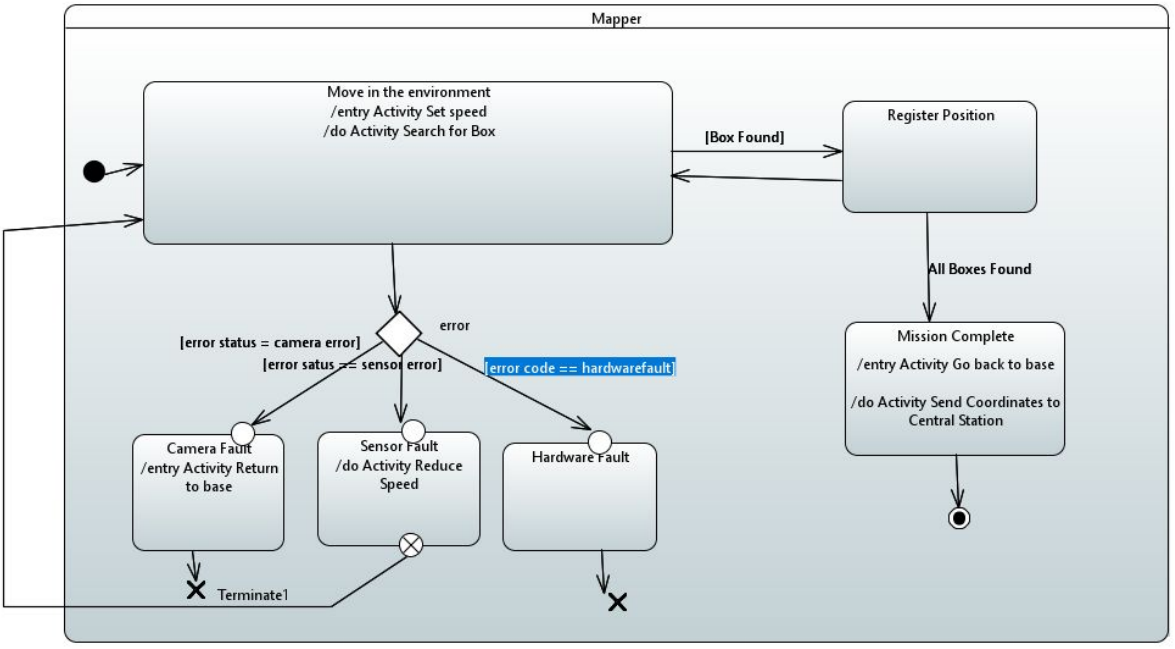
1. **Sensor Fault**: if the number's length is 3 or 4, we simulate a broken sensor. In this case, the robot will lower its speed and proceed more carefully. At the same time, we keep track of the number of broken sensors. (We did not implement the case in which all sensors break down, but it can be reconducted to a Severe Hardware Fault)
2. **Camera Fault**: if the number´s length is 5, we simulate a camera issue. The robot will then the robot will act as if a Severe Hardware Fault occurred.
3. **Severe Hardware Fault**: if the number´s length is 6, the robot will shut down and stop where it is.

To indicate the status of the robot, we also added a lamp actuator to every robot. If the lamp is on, the robot is working properly; if the lamp starts blinking, a minor fault happened (e.g. sensor fault); if the lamp shuts down, the robot is completely broken.

2. Objects internal behavior

Author(s): Fabiola Loffredi, Carmen Toyas Sánchez

- Class: **Mapper**



This State Machine Diagram represents the class **Mapper.**

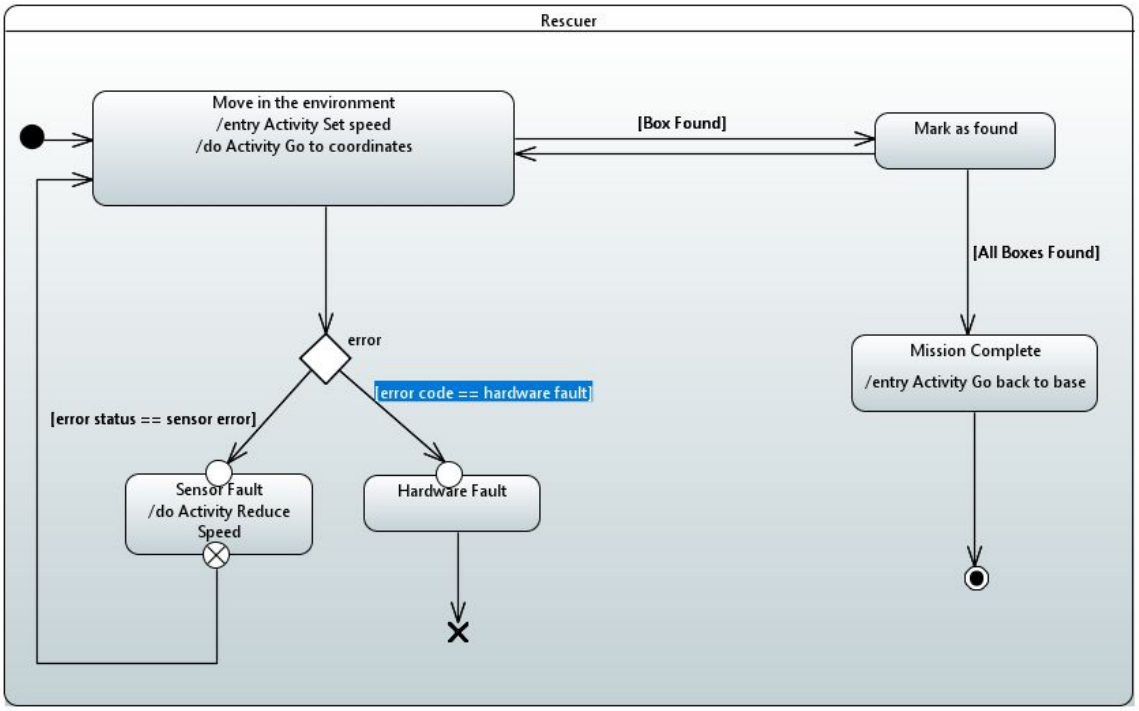
After the initial state, the system enters the state '`Move in the environment`'. Here, the entry activity '`Set Speed`' is executed and then the do activity '`Search for Box`'.

Once the system finds a box, the guard [`Box found`] evaluates TRUE and the transition occurs, then the state '`Register Position`' is entered, and the entry Activity '`Update Array`' is executed. The state is then exited and the system returns to '`Move in the environment`'; this sequence executes until all boxes have been found. When this is the case, the guard [`All Boxes Found`] evaluates to TRUE, then the transition occurs and the state '`Mission complete`' is entered. [write according to change] once this is done, the final state is reached and the execution stops.

In our system, while the Mapper robot is searching for the boxes, an internal error may occur. In the state machine diagram, we represented this event as well. If an error occurs, depending on the random error that can occur, a different state is entered. There are three possible cases:

1. if [`error status == camera error`]: the state '`Camera Fault`' is entered and the system is terminated.
2. if [`error status == sensor error`]: the state 'Sensor Fault' is entered, the do activity Reduce speed is executed and the system state is returned to move in the environment
3. if [`error code == hardware fault`]: the state '`Hardware Fault`' is entered and the system is terminated.

- Class: **Rescuer**



This State Machine Diagram represents the class **Rescuer.**

After the initial state, the system enters the state '`Move in the environment`'. Here, the entry activity '`Set Speed`' is executed and then the do activity '`Go to coordinates`'.

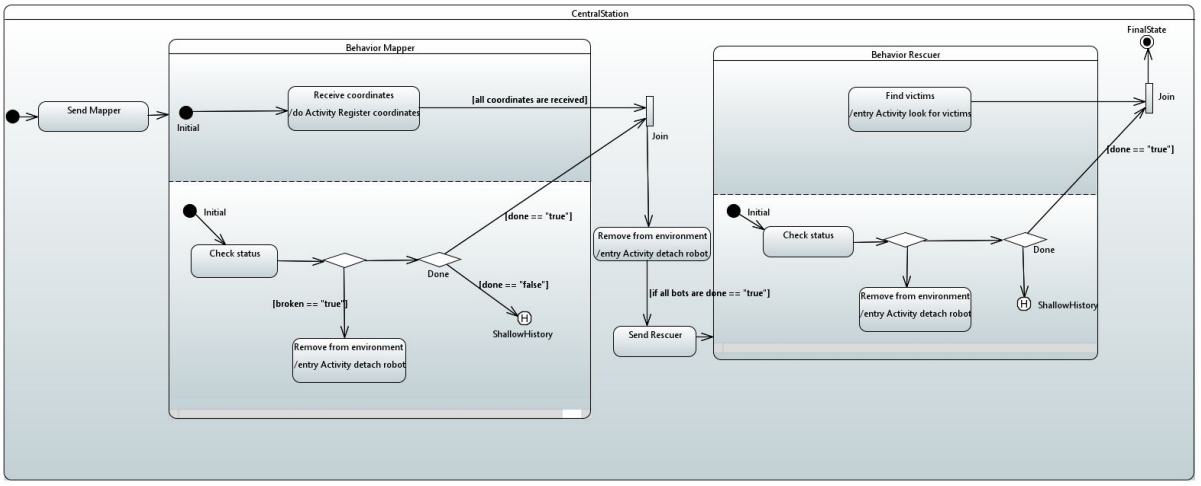
Once the system finds a box, the guard [`Box found`] evaluates TRUE and the transition occurs, then the state '`Mark as found`' is entered and the entry Activity '`Update Array`' is executed. The state is then exited and the system returns to '`Move in the environment`'; this sequence occurs until all boxes have been found. When this is the case, the guard [`All Boxes Found`] evaluates to TRUE, then the transition occurs and the state '`Mission complete`' is entered. [write according to change] once this is done, the final state is reached and the execution stops.

In our system, while the Rescuer robot is searching the coordinates, an internal error may occur. In the state machine diagram, we represented this event as well. If an error occurs, depending on the error, a different state is entered.

There are two possible cases:

1. if [`error status == sensor error`]: the state '`Sensor Fault`' is entered, the do activity '`Reduce speed`' is executed and the system state is returned to move in the environment
2. if [`error code == hardware fault`]: the state '`Hardware Fault`' is entered and the system is terminated

- Class: **CentralStation**



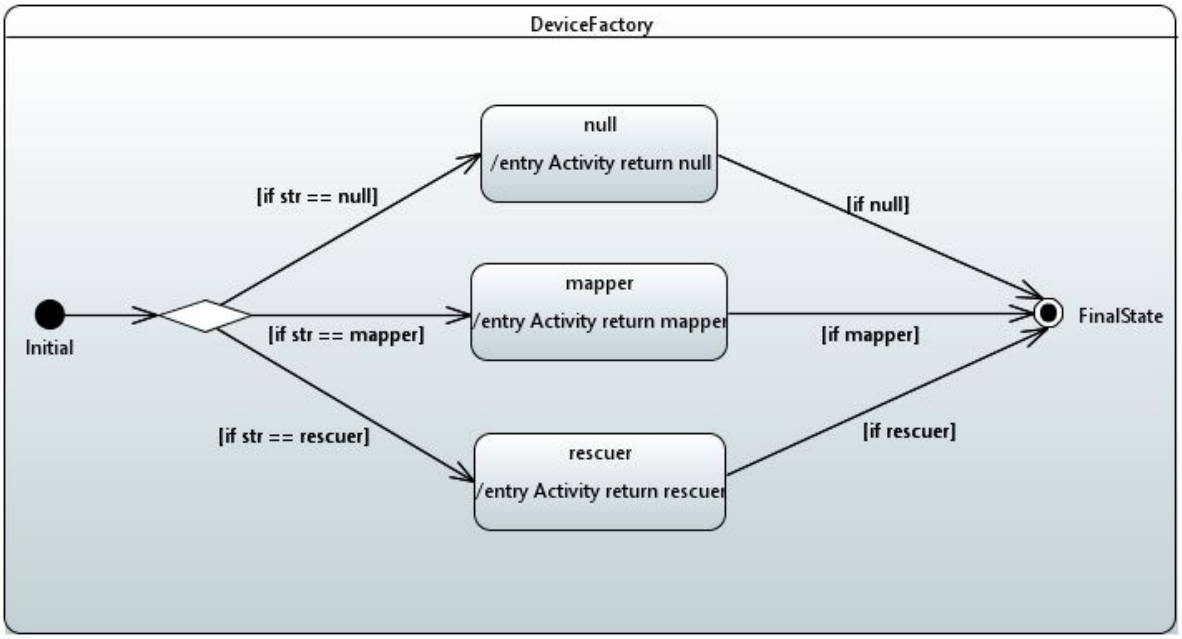
This State Machine Diagram represents the class **CentralStation.**

After the initial state, the system enters the state '`Send Mapper`', which represents the moment when the Central Station spawns a robot of type Mapper in the environment.

After this, the substate '`Behavior Mapper`' is entered. This state is an orthogonal state where two different activities occur at the same time. In the first region, after the initial state, the system enters the state '`Receive Coordinates`' and the do activity '`Register Coordinates`' is executed. At the same time, in the second region, after the initial state, the state '`Check status`' is entered. Here, there are two guards that are always checked: the guard `[Broken]`, which describes the moment when a hardware error occurs in the robot. If this is the case, then the guard will evaluate to TRUE, the transition will occur, the status '`Remove from environment`' will be entered and the entry activity '`Detach robot`' will be executed (the robot will be removed from the environment). Then, the other guard `[Done]`, which checks whether or not a Mapper has found all of the boxes. If the guard evaluates to FALSE, then the transition to the '`History State`' and the whole process is repeated until the guard evaluates to TRUE. Unless the `[Broken]` guard evaluates to TRUE, when both the `[Done]` and the `[all coordinates are received]`, the substate is exited, the transition is executed, and the state '`Remove from Environment`' is entered. Here, the entry activity '`detach robot`' is executed. When all the robots are removed from the environment, the guard `[all bots are done]`, the transition occurs, and the system enters the status '`Send Rescuer`', the entry activity '`add Rescuer`' is executed and the substate '`Behavior Rescuer`' is entered. Again, this substate is an orthogonal state, similar to the previous one: the "status control" is the same as for in '`Behavior Mapper`'. However, in the first region, we have a different state '`Find Victims`'; nevertheless, these transitions have the same mechanics of the previous one.

Once the two regions are done with the execution, the system enters the final state.

- Class: **DeviceFactory**



This State Machine Diagram represents the class **DeviceFactory.**

After the initial state, the system can assume one of three different states depending on how the guard will evaluate:

1. [`if str == null`]: if this guard evaluates to TRUE, the transition will occur and the state '`null`' is entered. Here, the entry activity '`return null`' is executed, then the next transition takes place and the system reaches the final state.
   This state represents an eventual error handling in the system.

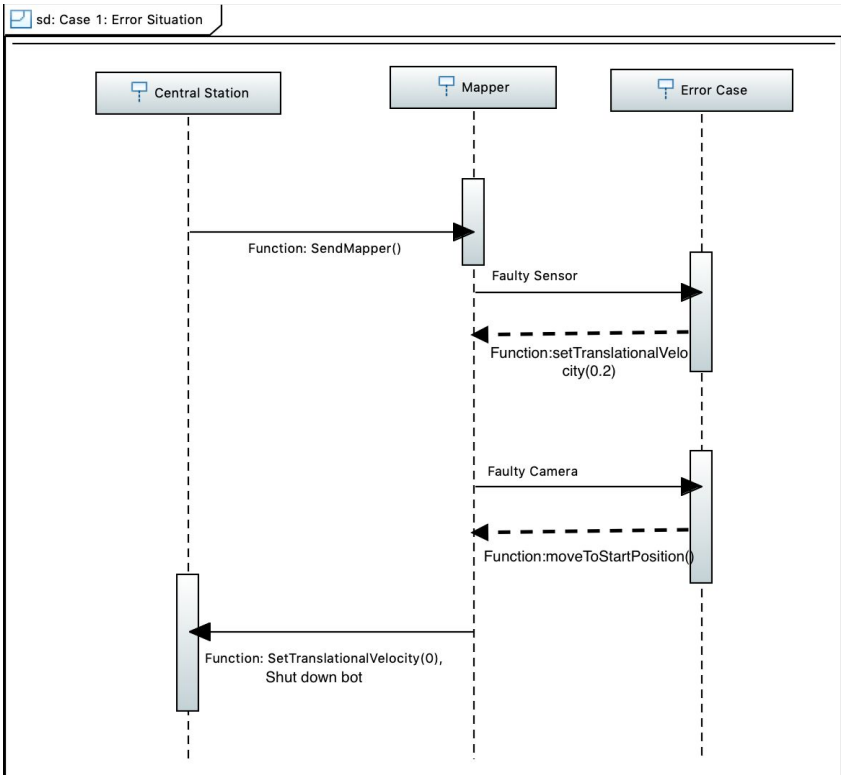
2. [`if str == mapper`]: if this guard evaluates to TRUE, the transition will occur and the state '`mapper`' is entered. Here, the entry activity '`return mapper`' is executed, then the next transition takes place and the system reaches the final state.
   This state represents when the robot of type Mapper is generated.

3. [`if str == rescuer`]: if this guard evaluates to TRUE, the transition will occur and the state '`rescuer`' is entered. Here, the entry activity '`return rescuer`' is executed, then the next transition takes place and the system reaches the final state.
   This state represents when the robot of type Rescuer is generated.

3. Interactions

Author(s): Anna Claire Favié and Roshini Ramasamy

Sequence diagram situation 1: Error cases



When the robot is set into the environment we allow it to operate by a function called setMode(), we give it the abilities to move around, avoid obstacles and in case of when the robot is a mapper to find victims.

We also give the robots a function where it can avoid obstacles so it will keep a 0,3 meters distance from the objects. Then it will determine what kind of object it is.If all this works then the robot was successfully created.

When the robot is a mapper it will inspect the environment and send coordinates of any victims found to the central station, if it is a rescuer the data sent to the central station by the mapper will then be send to the rescuer, so it can full-fill its job. The central station jobs is to receive and save data in order for the robots to access it.

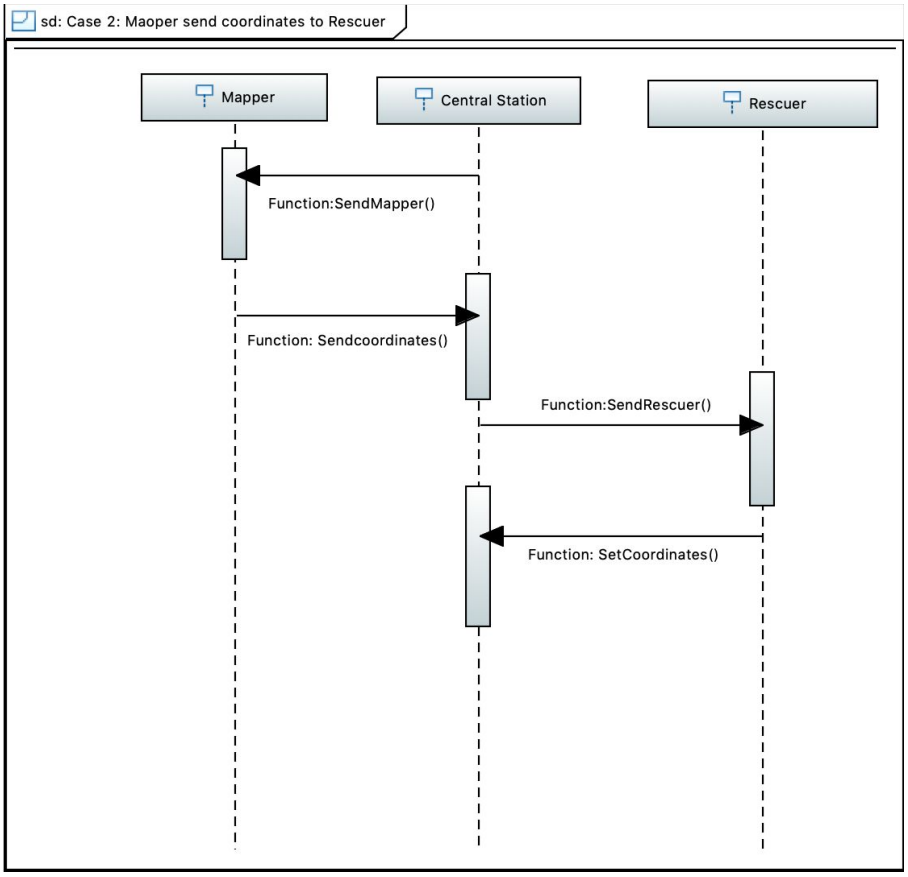
Now we will cover the case when there happens to be a hardware fault with one of the bots. Both bots contain the same hardware, essentially sensors, a camera and sonar. As mentioned in one of the previous sections, we have handled the case for 3 different hardware faults using a random number method, each resulting in a different outcome.

The first case is when there is a fault with one of the sensors, we have decided to slow the bot down in the case where there is a victim present but as one of the sensors is not functional, it may have overlooked the victim. The slower speed allows more time for the function of the bot to work. The next case is when the camera has stopped working, this is a crucial part of the bot and plays a big role in identifying the victims as well as distinguishing them from walls and other objects. Since it plays such an important role, the bot is sent back to its start position. Finally, when there is

a big hardware fault such as the robot not being able to read the coordinates, the bots speed is brought to 0 and it is turned off.

With all errors cases, there is an error message printed out referring to the specific case so the user is aware. Another way that the user can be notified that there is a malfunctioning bot is by looking at the lamp actuator that has been added which was specified in section 1.

Sequence diagram situation 2: Moment mapper sends the coordinates to the rescuers



In the situation where the mapper sends the coordinates to the rescuers, there is a whole interaction that happens before the actual coordinates are sent to the rescuer.

We make use of 2 robots: mapper and rescuer. The mapper is first in the game, it will move freely around in the environment and avoid obstacles autonomously. The mapper is sent into the environment by the function sendmapper(). The rovers goal is to locate the boxes in the field and determine whether it is a box or a wall. In case of a wall, the robot will rotate 5 degrees, until it does not find an obstacle, and then will continue its search.

In order to make sure the coordinates are correct we need to make sure that the coordinates being sent by the mapper are in type Point3d and of course if the coordinates are from a box and not of a wall.

Our robots are able to distinguish a box from a wall, it will be at least 0.3 meters away from the obstacle in question, the robot will move around it and if the camera detects an ongoing object it means it is a wall, if the surface isn't ongoing the robot detects it as a box.

When a box is identified, its coordinates will be saved in the central station to make

sure no box can be discovered twice as well as that the data is necessary for the rescuer which he will be able to access the data via the central station. The central station is the observer, so it will expect the coordinates of the objectives from the mappers. After doing their part, the mapper will start all over again and search further.

So the coordinates (in type Point3d) recovered by the mapper will be sent to the central station by the function setcoordinates(), this function is used to retrieve the coordinates from the mapper.

Mapper is a subclass of robot, it has several attributes that are important to find the coordinates:

camerasensor: represents the camera used to search for objectives in the environment

bufferedImage: this is a variable that stores the current image recorded by the camera

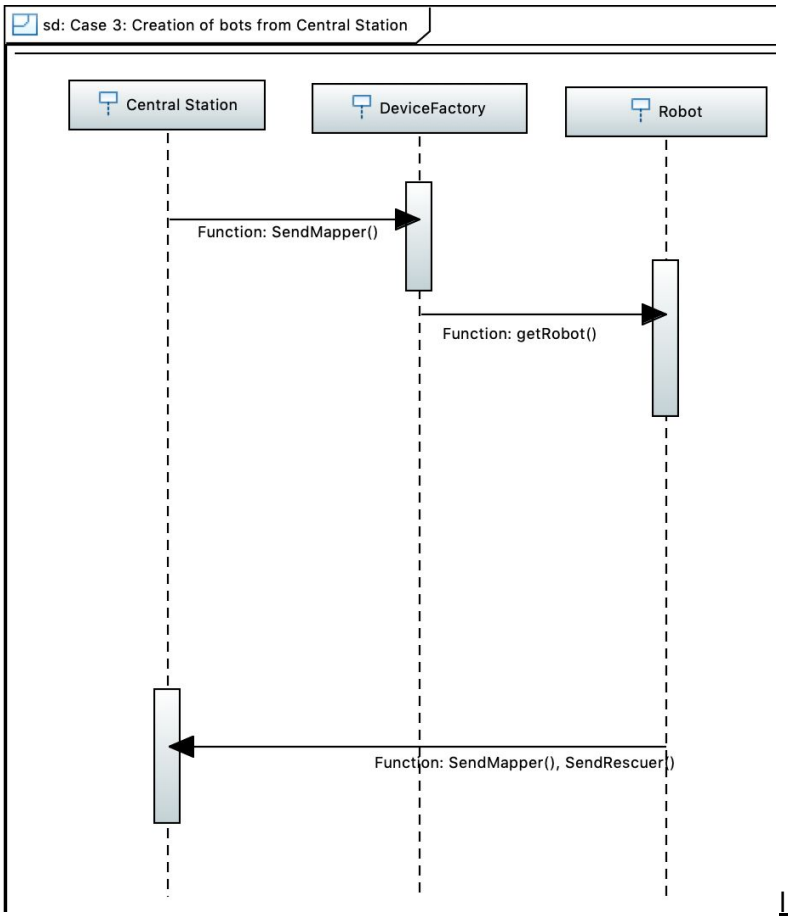
sonars: represents the instances for sonars of type RangeSensorBelt.

Once central station receives the coordinates it saves them and will use the function sendrescuer() to send a robot of type rescuer into the environnement. The rescuer will retrieve the objectives. The rescue robots are robots that are affixed to the objectives.

So by using setCoordinates(), the central station will store the coordinates received from the mapper and therefore this data will be available to the rescuers which will now be able to locate the objectives and retrieve them.

We use the function: missioncomplete() when the robot mapper has found all the objectives. Therefore, all the coordinates of the objectives will be or have been sent to the central station which gives it complete knowledge.

Sequence diagram situation 3: Creating bots from Central Station using DeviceFactory



In this situation, we are covering the case where the signal to create bots is sent from the central station. The Central station calls device factory to create the robots needed for the mission. The DeviceFactory is where the robots are created, it manages two different types of robots, mappers and rescuers. The mappers are the first bots that are created and then are followed by rescuers.

The class DeviceFactory is the one used once the environment is created and follows the instruction of the central station, this class can exist without the central station but the bots will not be created. Device factory is called to generate both the mappers and rescuers, it does so by using the method getRobot which returns a generic bot with no specification of its type.

Within getRobot, this operation takes place using by taking a string parameter for the function. This string allows for the specification of whether a Mapper or rescuer is created and the method creates subclasses of the class Robot.

For example when the environment is first created, the area needs to be scanned and any victims must be identified for this, The sendMapper function is called from main, which contains the string ´mappers´ as its parameter, it then calls the DeviceFactory class through the central station and creates a mapper bot which is then placed in the field.

4. Implementation remarks

Author(s): Cristiano Milanese

Strategy:

Once the code has been generated from the UML diagrams all the methods have been implemented with the chosen algorithms, each state is implemented through a string variable that set the current mode of the rovers, each state is associated with a behaviour that is expressed by the robot.

Key solutions:

Randomly walking around is the way mappers experience the environment, casually encountering red boxes with their camera and move towards them to register the closest position possible. The camera is mounted on each of the mappers and tests the central pixel of each frame, assessing its RGB values. If the returning value has any shade of red inside, the mode is switched to "found", consequently toggling the bot behavior. All positions are sent to Central Station, which conserves a list of all item located. Since 4 is the number of "victims" in the environment, as soon as the maintained lists contains 4 elements the mappers are considered done and the rescuer enters into play. For solution's sake our final implementation, Central Station teleports them at the indicated locations, pretending their are approaching the red box in order to "save the victims". All the bots are created at initiation time by Central Station and re-attached to the environment at due time. The mission is complete when the rescuer get back at their initial position and therefore all victims have been found.

Main Java class:

The main method is in a separate class called Main, which is responsible for the creation of the simbad environment and render the scene graph.

Link for YouTube video of the execution of our system in the SimBad simulator:

https://youtu.be/lrtP3uIUPhQ