# Software design
# Team project – Deliverable 3

**Team number**: 16

**Team members:**

| Name | Student Nr. | E-mail |
|---|---|---|
| Eliane Kadouch | 2623437 | e.r.kadouch@student.vu.nl |
| Karol Komorniczak | 2622762 | karol.komorniczak@gmail.com |
| Maciej Juzoń | 2618287 | maciek.juzon@gmail.com |
| Gaweł Jakimiak | 2617124 | gawel.jakimiak.ib@gmail.com |
| Jiyoung Oh | 2610337 | godinggool@gmail.com |

This document has a maximum length of 15 pages.

## Table of Contents

TABLE OF CONTENTS 2

1. THE NEW REQUIREMENT 3

2. OBJECTS INTERNAL BEHAVIOR 3

3. INTERACTIONS 4

4. IMPLEMENTATION REMARKS 4

5. REFERENCES 5

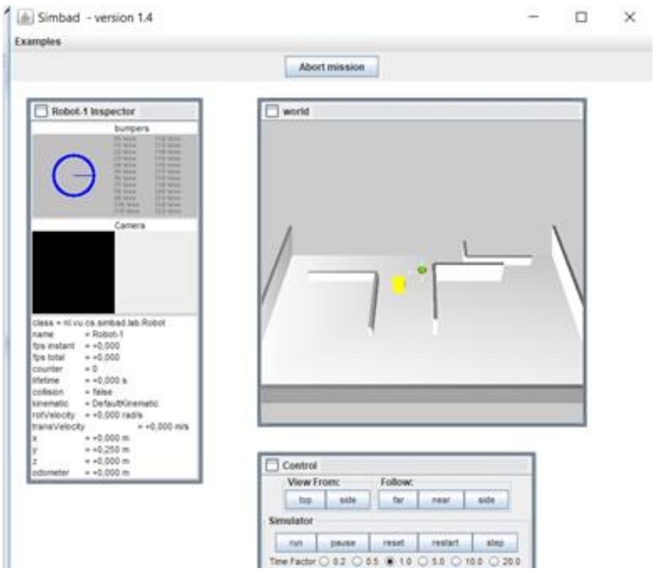
## 1. The new requirement

**Author(s)**: <Maciej Juzon, Gaweł Jakimiak>

Our group has been assigned the requirement change C, which states: "The system shall support the situation in which the operator pushes a "Stop-everything" button for immediately stopping the whole mission due to some severe circumstances..". However, adapting our model to this requirement was not that difficult, as our system from the very beginning was being designed with a feature that the user can stop the system whenever she or he wants.

As it can be seen in our previous deliverable, the use case diagram contained a use case "Stop the system", which was an extension to our "Start the system" use case. In order to implement the new requirement we first come up with the idea of adding a key listener to our Simbad JFrame, but after discussing the idea with other group members, we have decided to create new button and add it to our Simbad JFrame as in our opinion it looks more convincing and the user automatically can understand the purpose of the button, so he or she does not have to read through the manual to find out which of the keyboard keys is responsible for stopping the system.

In our system, the Simbad frame is instantiated in the main class and thus the whole process of adding an additional button to it is done in main. We create a JPanel to which we add our custom button and assign an action listener to it. With this simple solution, we can override the method actionPerformed, so whenever the button is pressed our code will be executed, which in this case is setting the state to 6 (responsible for ending the mission). Because of the fact that all this process is done in main, we did not have to alter our class diagram and the code required just few adjustments. Below Figure 1. shows our new custom Simbad control window, where the "Abort mission" button is positioned at the top of our Frame.



```
class = nl.vu.cs.simbad.lab.Robot
name          = Robot-1
fps instant   = +0,000
fps total     = +0,000
counter       = 0
lifetime      = +0,000 s
collision     = false
kinematic     = DefaultKinematic
rotVelocity   = +0,000 rad/s
transVelocity               = +0,000 m/s
x             = +0,000 m
y             = +0,250 m
z             = +0,000 m
odometer      = +0,000 m
```

Time Factor ○ 0.2 ○ 0.5 [✓ 1.0] ○ 5.0 ○ 10.0 ○ 20.0

Figure 1. Representation of our new Simbad control window

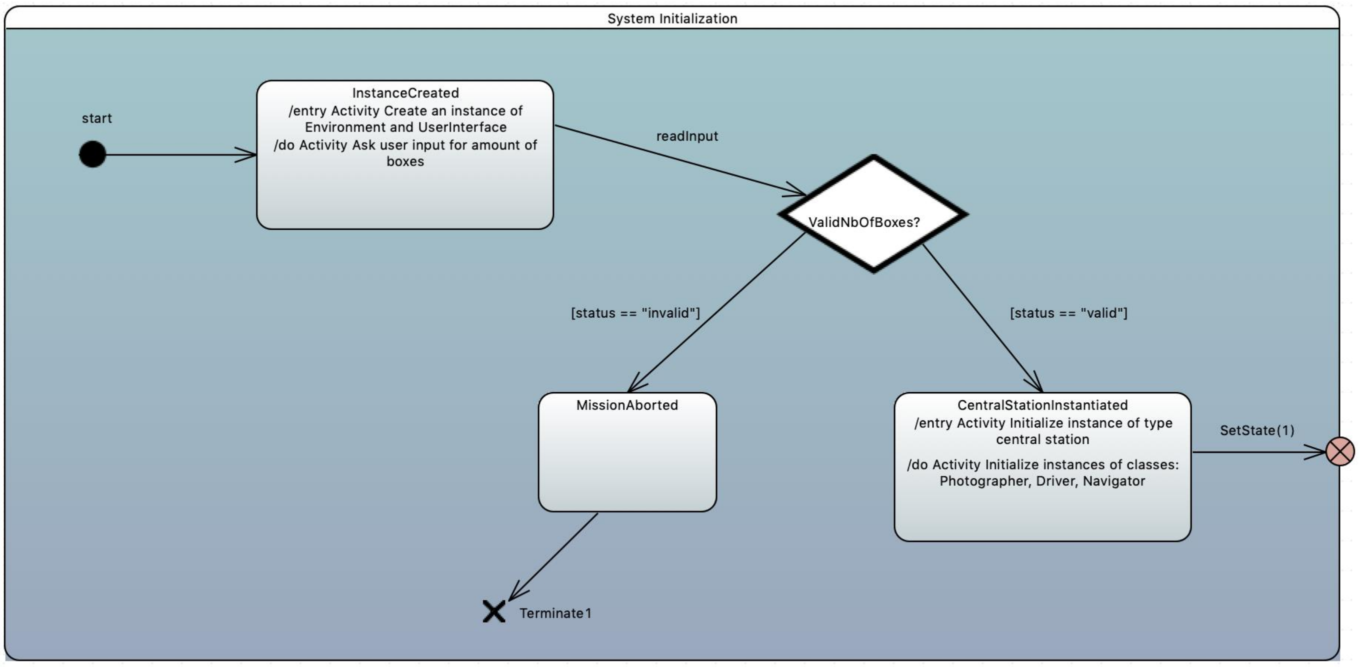
## 2. Objects internal behavior

**Author(s)**: <Eliane Kadouch, Jiyoung Oh>

As explained in the description, a state machine diagram needs to be precise enough to get a clear understanding of our system's internal behavior. To do so, we decided to separate the system into 3 big states, each represented by a diagram in order to avoid unclear representations and make it easier for the reader to understand. The first diagram will describe the system initialization, the second one will focus on the behavior of Central Station, Environment, Observers, and the last one will illustrate how the Photographer, Navigator and Driver interact with each other.

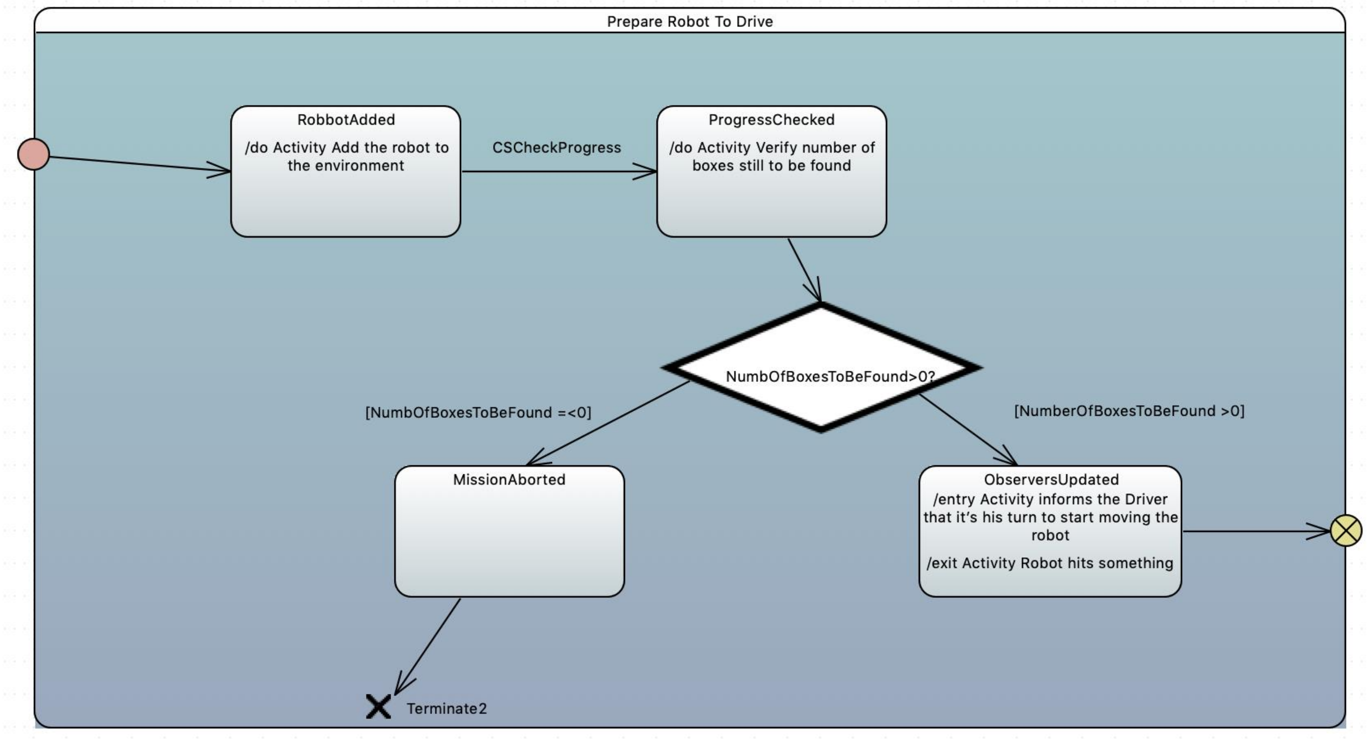
*Diagram starts at state 0

### ***System Initialization***


System Initialization
start
InstanceCreated
/entry Activity Create an instance of Environment and UserInterface
/do Activity Ask user input for amount of boxes
readInput
ValidNbOfBoxes?
[status == "invalid"]
[status == "valid"]
MissionAborted
CentralStationInstantiated
/entry Activity Initialize instance of type central station
/do Activity Initialize instances of classes: Photographer, Driver, Navigator
SetState(1)
Terminate1

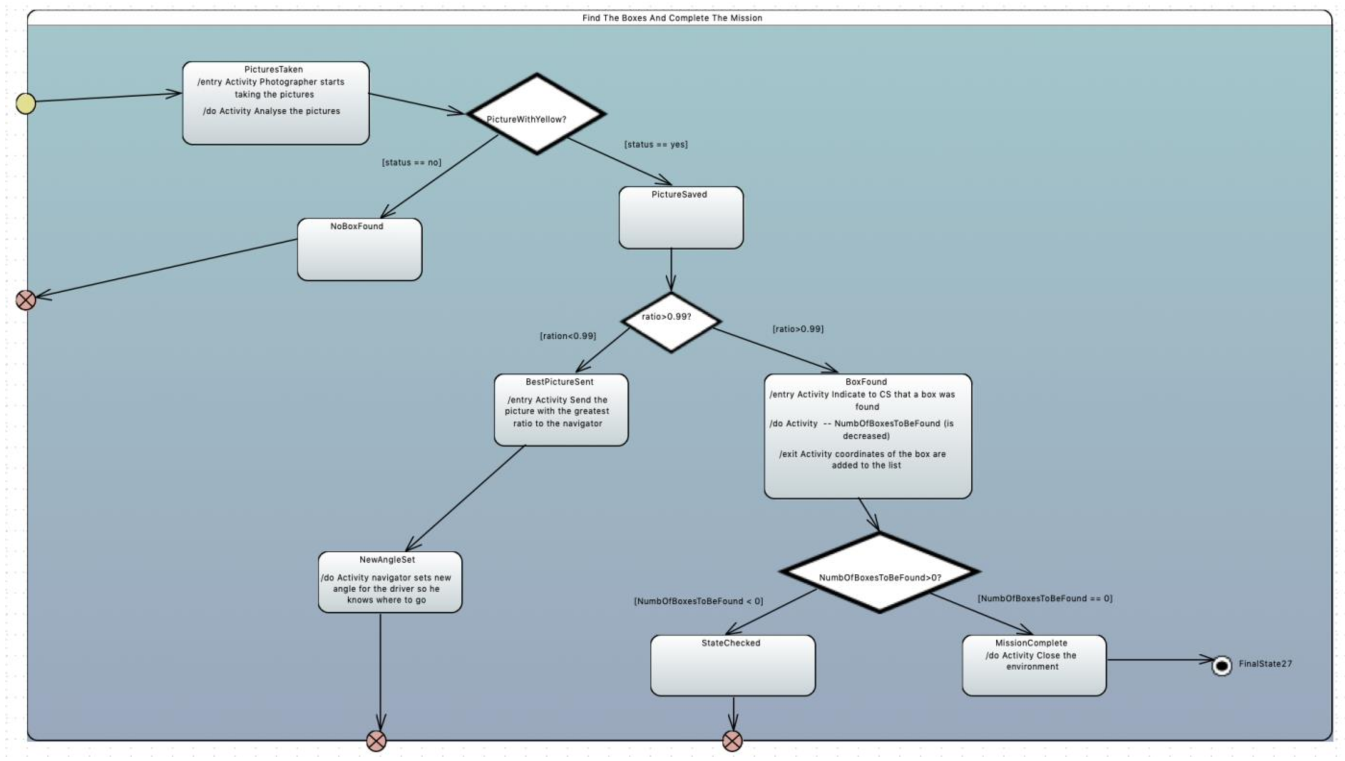
The moment our system is started an instance of Environment and UserInterface are created. UserInterface is then used to ask user about the number of boxes to be added to the environment, together with their coordinates. After User inputs all necessary information, the system checks if the number of boxes are valid, in other words if it is less or equal to 0 with a choice pseudo-state, shown as a diamond with one transition arriving and two transitions leaving. If it is less or equal to 0, the number of boxes to be found is invalid which aborts the mission and terminatesthe system with a cross, indicating that the lifeline of the state machine has ended. After verification, all Boxes are added to the environment and their number is saved. Then an instance of CentralStation is instantiated which also creates an instance of Driver, Observer and Navigator and assign them to a single Robot. State is changed to 1.

## ***Prepare robot to drive***


Prepare Robot To Drive
RobbotAdded
/do Activity Add the robot to the environment
CSCheckProgress
ProgressChecked
/do Activity Verify number of boxes still to be found
NumbOfBoxesToBeFound>0?
[NumbOfBoxesToBeFound =<0]
[NumberOfBoxesToBeFound >0]
MissionAborted
ObserversUpdated
/entry Activity informs the Driver that it's his turn to start moving the robot
/exit Activity Robot hits something
Terminate2

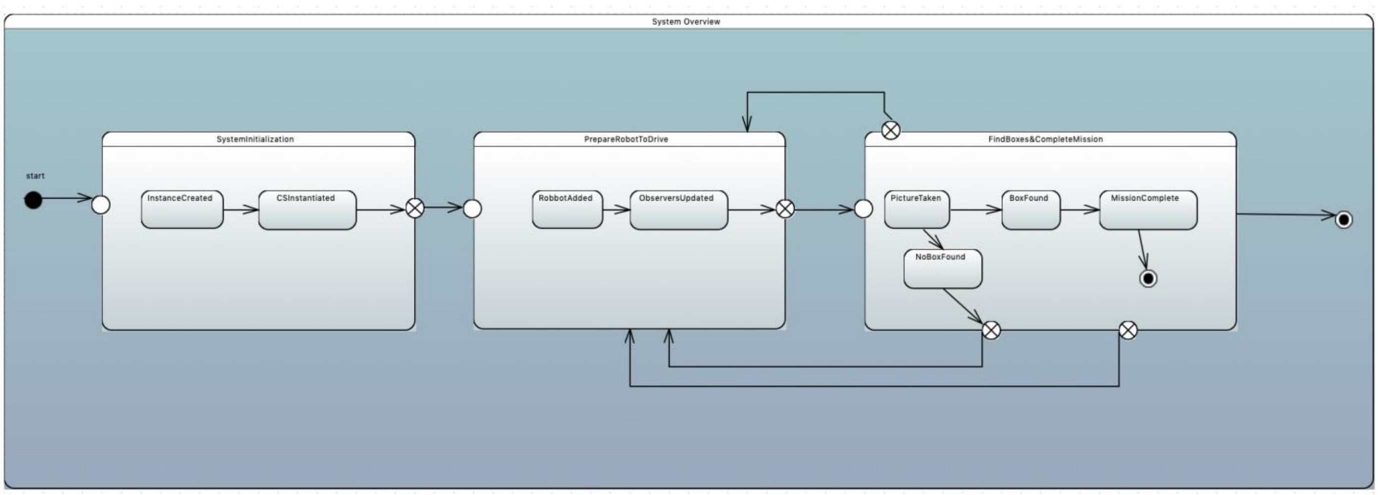
Now CentralStation checks the number of boxes and adds robots to the environment using the addRobot() method. The central station verifies the mission progress and checks the number of boxes still to be found. If it's bigger than 0 it updates all observers and informs the Driver that it's his turn to start moving the robot. If the number of boxes is less than 0, the Environment sends a message to the Central Station and the Mission is directly aborted which terminate the system with a cross, indicating that the lifeline of the state machine has ended. If not, it updates all observers and informs the Driver that it's his turn to start moving the robot. When it hits something it sets state to 2.

## ***<u>Find the boxes and complete the mission</u>***


Find The Boxes And Complete The Mission
PicturesTaken
/entry Activity Photographer starts taking the pictures
/do Activity Analyse the pictures
PictureWithYellow?
[status == no]
[status == yes]
NoBoxFound
PictureSaved
ratio>0.99?
[ration<0.99]
[ratio>0.99]
BestPictureSent
/entry Activity Send the picture with the greatest ratio to the navigator
BoxFound
/entry Activity Indicate to CS that a box was found
/do Activity -- NumbOfBoxesToBeFound (is decreased)
/exit Activity coordinates of the box are added to the list
NewAngleSet
/do Activity navigator sets new angle for the driver so he knows where to go
NumbOfBoxesToBeFound>0?
[NumbOfBoxesToBeFound < 0]
[NumbOfBoxesToBeFound == 0]
StateChecked
MissionComplete
/do Activity Close the environment
FinalState27

At state 2 Photographer starts taking the pictures and analyse them. If he finds a photo with yellow color in it it saves it's number. If not, it means that no boxes were found for the moment and the driver needs to navigate again, in other words, there is a transition to state 2. If the ratio of color yellow over other colors is higher than 0.45 it indicates to central station that the box was found. NumbOfBoxesToBeFound is decreased and coordinates of the box are added to the list. If the photographer was unable to find a photo with yellow to non-yellow ratio greater than 0.45, he sends the best photo (best ratio) number to the navigator. If it did not find a photo a transition to exit point is made and the sets state back to 1. After that, the navigator sets new angle for the driver so he knows where to go. The process is repeated until all boxes are found.

## ***System Overview***

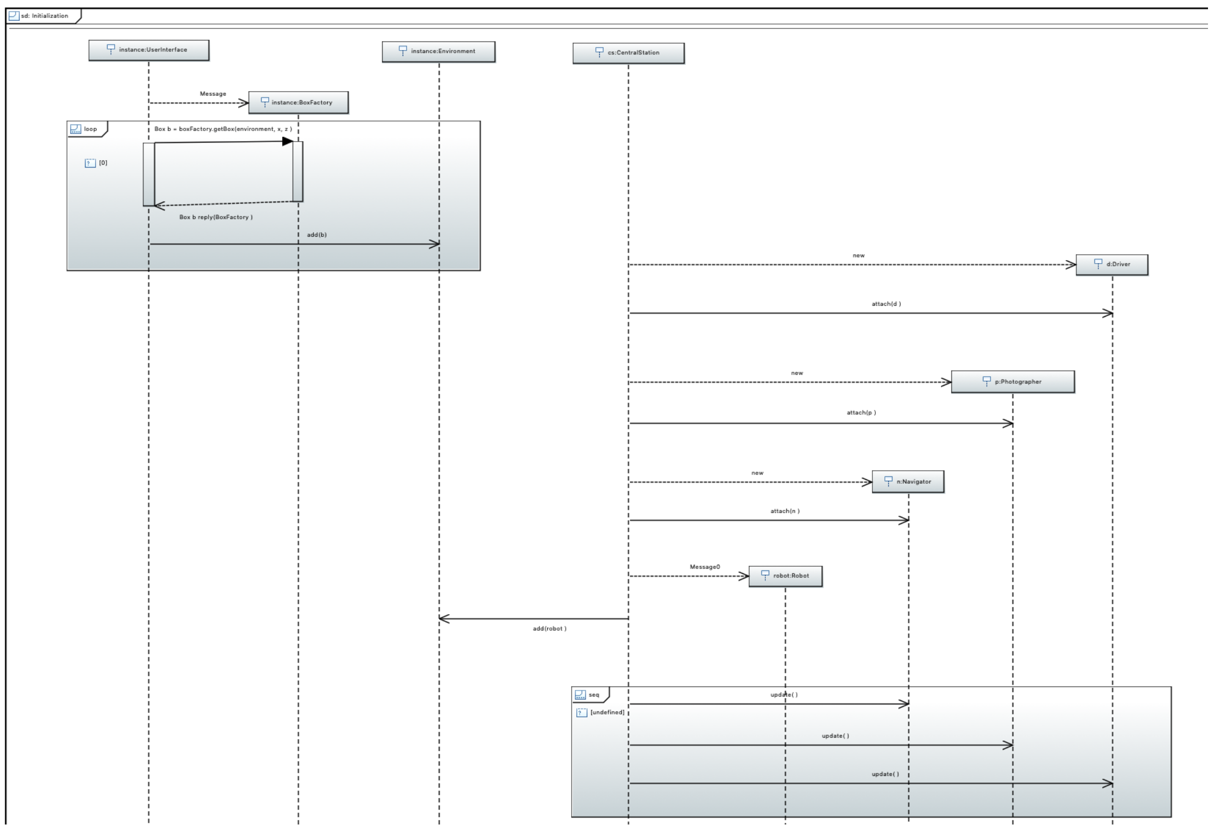


This section is meant to visualize all the above-mentioned state machine diagrams as one. It also adds on additional interactions between the second and the third diagram. In the third diagram when no box has been found, the system reverts back to the second state and proceeds with searching for the box. The two other exit points (one going from the bottom, one from the top) signify the exit points from the third diagram and show their exact destination. In diagram 3 when a box has been found but was not the last one or state 4 has been initiated (where a trace of yellow has been found and the robot drives towards it), the state machine exits to the second state machine where the robot continues to move and search for an object. We can notice a trend of consecutiveness, meaning that the state machines are interconnected follow each other from the initial to the final node.

## 3. Interactions

**Author(s)**: <Karol Komorniczak, Jiyoung Oh>

### ***System Initialization***

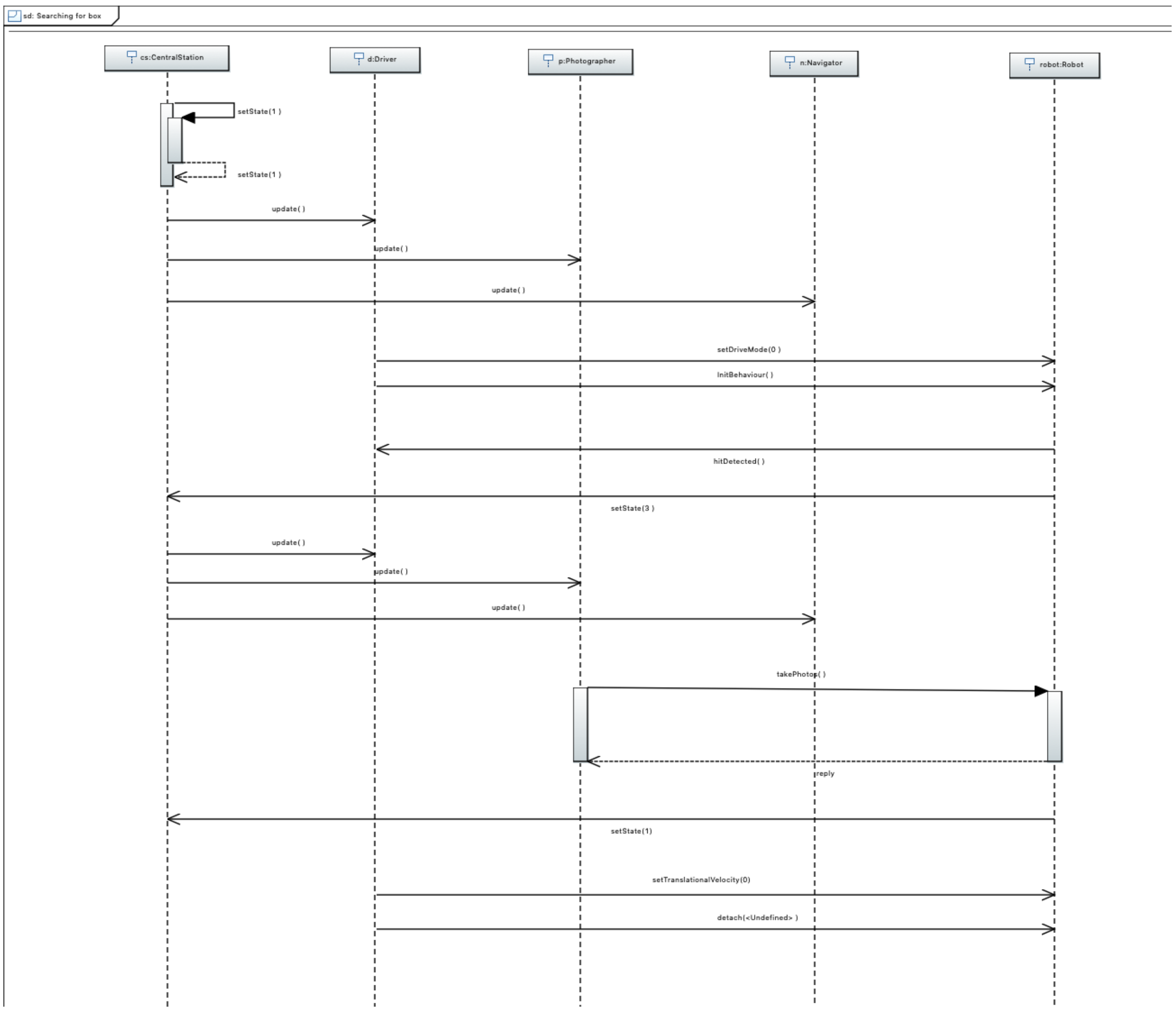


The first diagram showcases what happens at the start of the system. There are 4 Classes present in the diagram that are responsible for creating the basis for the system and producing additional objects in the process. After the environment has been initialized, the boxes need to be added to the environment. The first interaction begins with the creation of Class "Box factory", responsible for putting boxes on the map. The instance of this class is created by the user interface. The user interface then sends a synchronous message to the newly created BoxFactory and requests objects of the class box, which are then put directly into the environment by the UserInterface. The whole process of requesting a Box object and putting it into the environment happens in a loop, which iteration count is determined by the number of boxes the user requests to put on the map at the start of the system. Since the sequence diagram does not cover the exterior relations between the user and the system, we assume that there is one box on the map, therefore the loop executes once in this case.

Once the boxes are laid out onto the environment, there is now a need to place the robot into the environment. The central station takes care of this task. First, it creates instances of classes steering and maneuvering the robot, so instances of classes: Navigator, Photographer, and Driver. Our system has implemented the Subject observer design pattern, therefore after creation, each of the created class is linked with the “attach()” method to the Central station object, which extends the Subject class. The Subject class is responsible for updating each of the Observers about the state of the system. Right after that, the Robot object is also created by the central station and added to the environment by it.

Next the state is changed indicating that our system is in state 0 and has been fully prepared for the process of box searching. The activation of the method causes the central station class to update all the observers about the current state, indicated by the three update messages coming from the central station to all the observers. We presented the updating process in the combined parallel fragment to signify that order of the different operands is irrelevant, because the observers may be updated in any order. At this point, the search for the box starts.

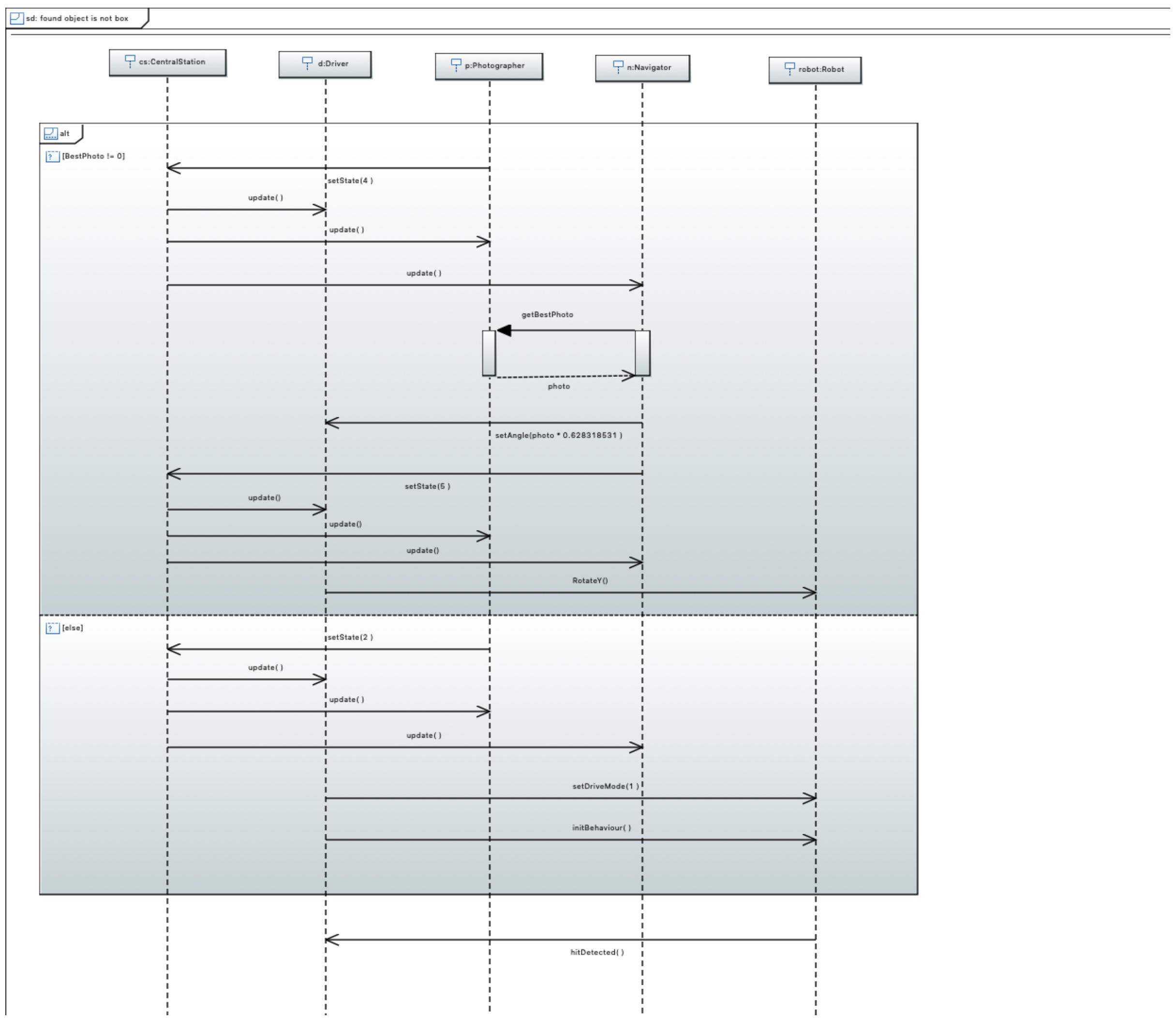
## ***Searching one box with instant success***



The next diagram presents an instance when one box was placed in the environment and is automatically found by the rover. We chose this simple case to show how objects communicate with each other when searching for the box without over complicating the process. More complex behavior options will be explained in the latter diagram.

This diagram picks up right after where the last diagram ended. After the state has been switched to 0, it is automatically switched to state 1 and then 2 using the if statements in the central station. We decided to show setstate(1) as a message to self in the central station class as it is a part essential in initiating the search as it tells the robot and the observers to start working and it is very important to show it as the behavior of the system is very closely related to the notion of states. Right after that, the central station updates every observer about the state. Each observer has a different corresponding task for each state. In this case, we are at state 2, so the Driver communicates with the robot to set its drive mode and sets its behavior, which in this case is moving randomly around the map. The initBehaviour() method is responsible for initiating the behavior in the robot class, which is executed every 20 seconds. While moving, if the Robot hits something, it reports it to the Driver using the "hitDetected()" and the method and changes the state to 3. Naturally, with the change of state, all Observers are updated by the central station. An array of photos  in order to determine if the hit object is an obstacle or a goal. This is where the Photographer Class comes in, activated at state 3. It asks the robot to for a photo of the encountered object and the robot object returns the photo for further processing. Because we assume that the robot finds the box at the first encounter, the photographer class determines that the box has been found and request the Central station to Change the state to 1 and conclude the search.

## ***The encountered object is not Box***



The last diagram shows the alternative scenario for when the robot has hit an object, but that object is not a box. This interaction would appear in the previous diagram if the photographer decided that the photo received from the robot is not sufficient to conclude the search. Two outcomes may happen in this case, which we modeled using an alternative combined fragment. We assume that after any of these cases execute the box will be found at the next object collision. The two cases are:

1. [BestPhoto != 0] means that the taken photo contains traces of yellow in it, which is the color of the box we are searching for. If there are indeed traces of yellow in the picture, the system switches to state 4 by the photographer, updating all the observers through the central station. The navigator observer then takes over and request the picture from the photographer class using the "getBestPhoto()" method, which it receives with a synchronous message. The navigator takes the photo and calculates the angle for the robot to drive in the direction of the noticed

color yellow from the picture. The angle value is then the passed over to the Driver by the Navigator using the “setAngle()” method. Then state is then set to 5 by the Navigator, which is passed to the central station class that then naturally updates every observer object. State 5 causes the Driver to activate a special method that is responsible for guiding the robot in the direction of yellow, therefore the driver rotates the robot with the newly calculated angle using the “rotateY()” method.

1. [else] If the photograph does not contain any yellow, the system switches into state 2, which means that the robot will continue moving randomly around the map. Therefore the driver orders the robot object to initiate such actions using the “setDriveMode()” and “InitBehaviour” messages, which we mentioned in the the first diagram.

After any of the two cases happen, the robot will be naturally moving around the map with a driving style dependent on which case just executed. Next interaction that will occur will be a “hitDetection()”, which is applicable to both cases, therefore not shown in the combined fragment. The diagram then follows the exact order of interactions as the second diagram when the “hitDetection()” method message is sent. Because we assumed that the objects will be found right after the occurrence of any of the cases, this diagram will completely resemble the second diagram after the mentioned message. We are mentioning this as it modeling the same elements of a diagram twice would overcomplicate the look of the last diagram.

## 4. Implementation remarks

**Author(s)**: <Maciej Juzon, Gawel Jakimiak>

In order to move from our UML models to the implementation process we first generated the code, using one of the functionalities of papyrus designer, which allows the user to generate the code based on our previously created class diagram. When we decided that our model is complete and finished, we generated the code. This created a separate folder with all our classes and interfaces in the workspace. However, the generated code, in the beginning, was incompatible with the simbad framework and it required minor changes.

First of all, the libraries required for displaying the robot simulation were not included in the folder created by papyrus designer and thus we needed to move all our classes to one of the folders used in the beginning for practicing the principles of simbad simulator and change the packages of our classes to proper ones. Secondly, as we were not obliged to include generalization classes provided by simbad environment, like EnvironmentDescription or Agent, in our class diagram, we had to manually add required extensions to given classes (Robot and Environment). After that our code was nearly ready for implementation of each class, we just removed couple of things related to our root element (for example we removed the D2 from D2.CentralStation as it was redundant).

Since we have prepared all the classes for deliverable 2 and even implemented some functionalities of our system (Environment, UserInterface, BoxFactory – those classes were implemented for the purposes of previous deliverable). Now we just needed to add missing functionalities and put everything together basing on our new UML models, namely the sequence diagram and state machine diagram.  As described in previous deliverables, main part of our system is the central station and thus we implement it in the first place. Central Station main goal is to coordinate the job of Navigator, Driver and Photographer ,manage the exploration of the environment and monitor the progress of the mission (record the number of boxes found and keep track of their coordinates).

Next, we implemented all our observers and here we encountered first difficulties. In our system observers are responsible for controlling the robot. However, only one of them can be in control of the robot at given period of time, so here we had to implement the Subject-Observer pattern. We used states in order to determine which one of our observers will be in control of the robot and what it will do. When implementing these things, we found that our initial idea of five states will not be enough and thus we had to add another one, which resulted in some changes in our diagrams. Moreover, the implementation of our photographer class was quite demanding, as it is responsible for image classification in our system. We come up with the idea of classifying pictures captured by robot camera based on the ratio of good (yellow) and bad (non-yellow) pixels. This allowed us to determine if the encountered obstacle is our box or not. We also used image classification for the purpose of navigating our robot. Every time our robot hit an obstacle the robot will it's camera in order to take a series of photos when it will be rotating around its own axis. Then photographer will get the created array of images and process it. If none of them will be classified as box to be found, the photographer will chose the best one (the highest ratio of yellow to non-yellow pixels) and navigator will navigate the robot in that direction. If another obstacle will be encountered, the whole process will be repeated.

Afterwards, we focused ourselves on implementation of our Robot class. In our system it functions just as a tool, which is used by the observers, but it still needed to provide some key functionalities like collision detection and taking photos. Implementing the second functionality was a bit tricky, as we had to figure out how to take a photo from the robot's camera and how to store it. We decided to use of Simbad Eye functionalities, which first create an empty BufferedImage and then copy the view from the camera to it. Later, this picture will be added to ArrayList of BufferedImages and then send to photographer for the purpose of classification and further actions. When the Robot was done, we had to implement our data type, which in our system is represented by coordinate class. It stores the position of the box and is used in the end to show the position of the found boxes to the user.

Finally, we had to implement the additional requirement, which was described in detail in section 1. All the new code was added to our main java class, which beside being responsible for the "Abort mission" button is also responsible for creating the instance of Simbad frame and invoking the whole process of environment creation via the user interface. When we finished the implementation and the whole project we really understood the importance of proper design process during software development. We all agreed that our previous approach "Do something and let see what is going to happen" was wrong, as it creates a big mess and is extremely hard to coordinate when working as group. This approach led us to the point where we had to abandon our first idea, rethink it and then redesign. After that everything seemed to be much more convenient, we were able to split the work fairly among all the group members and finally the implementation part was done smoothly and without any unnecessary work.

Below we have added a link to the video representing our working system. As you can see, our robot starts from the centre of the map and moves randomly until it hit an obstacle. Then it make a series of photos, which will be processed by the photographer and decide in which direction the robot should go next. Next the robot hits the box and again take a series of photos and then photographer classifies it as our goal. In the end the coordinates are being displayed in the console and the mission is aborted. For the purpose of the video we have just added one box in order to show all the necessary functionalities of our system.

Link to video:

https://youtu.be/wiTcEj5ro7k

## 5. References