Software design
Team project – Deliverable 2

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

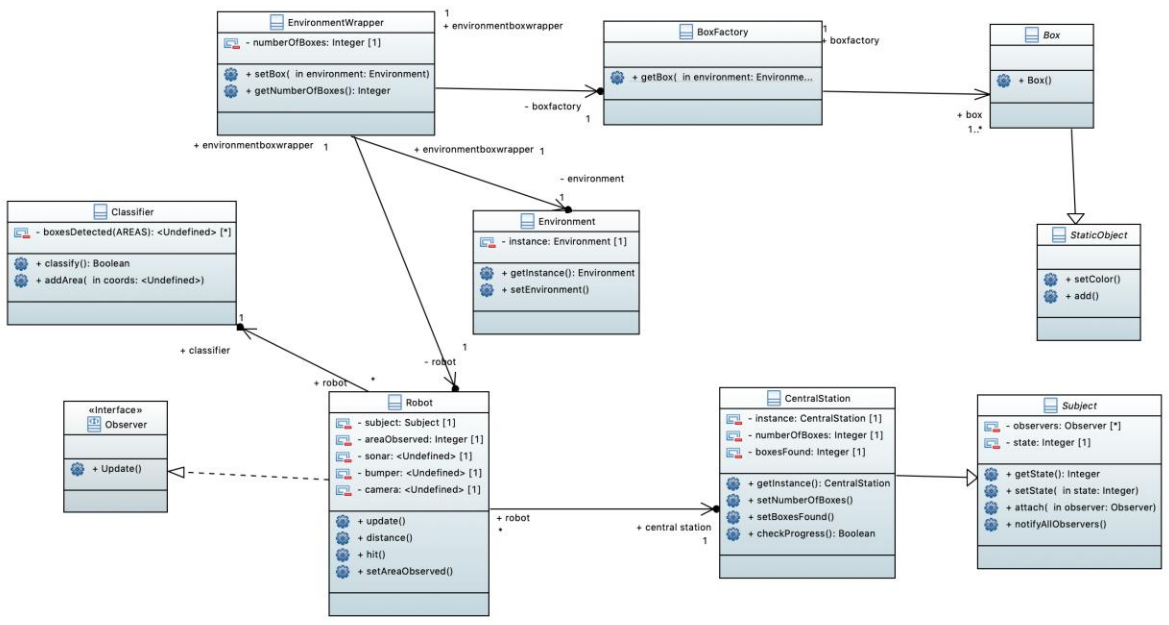
This document has a maximum length of 10 pages.

Table of Contents

1. Class diagram

Author(s): Gaweł Jakimiak, Maciej Juzon, Eliane Kadouch, Karol Komorniczak

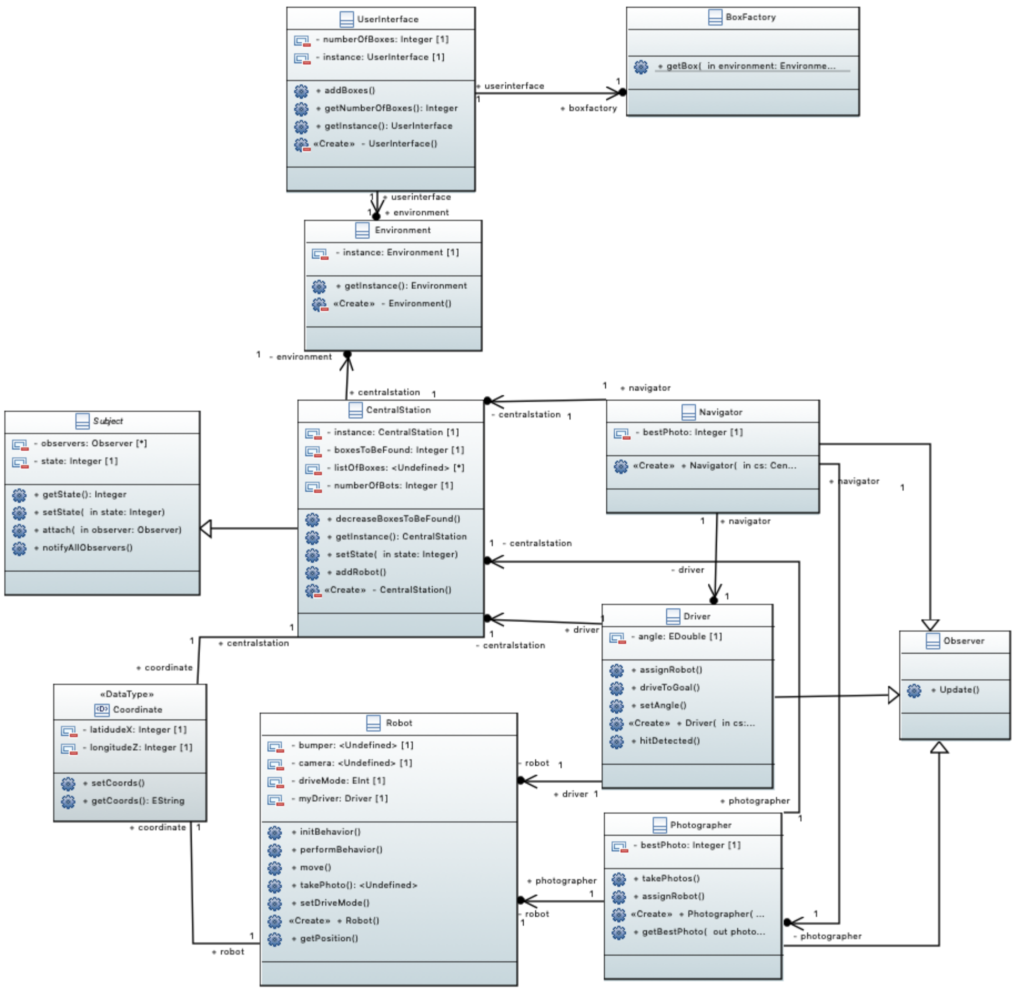
1. The idea and the diagram we have started with


EnvironmentWrapper
- numberOfBoxes: Integer [1]
+ setBox(in environment: Environment)
+ getNumberOfBoxes(): Integer
1
+ environmentboxwrapper
BoxFactory
+ getBox(in environment: Environme...
1
+ boxfactory
- boxfactory
1
Box
+ Box()
+ box
1..*
+ environmentboxwrapper 1
+ environmentboxwrapper 1
- environment
1
Classifier
- boxesDetected(AREAS): <Undefined> [*]
+ classify(): Boolean
+ addArea(in coords: <Undefined>)
Environment
- instance: Environment [1]
+ getInstance(): Environment
+ setEnvironment()
StaticObject
+ setColor()
+ add()
1
+ classifier
1
- robot
+ robot
*
«Interface»
Observer
+ Update()
Robot
- subject: Subject [1]
- areaObserved: Integer [1]
- sonar: <Undefined> [1]
- bumper: <Undefined> [1]
- camera: <Undefined> [1]
+ update()
+ distance()
+ hit()
+ setAreaObserved()
+ robot
*
+ central station
1
CentralStation
- instance: CentralStation [1]
- numberOfBoxes: Integer [1]
- boxesFound: Integer [1]
+ getInstance(): CentralStation
+ setNumberOfBoxes()
+ setBoxesFound()
+ checkProgress(): Boolean
Subject
- observers: Observer [*]
- state: Integer [1]
+ getState(): Integer
+ setState(in state: Integer)
+ attach(in observer: Observer)
+ notifyAllObservers()

Our first class diagram represented a very desperate and chaotic attempt to integrate as many design patterns as possible. Only after we have tried to implement the code we have generated, it slowly started to appear, that we have incorrectly understood most of the ideas. The biggest problem we have encountered was with Subject-Observer pattern, which for us seems vital for the whole project. At the beginning we decided that our *environment* will be searched by four *robots* of the same instance. We wanted to distribute the area of the environment between them using the **CentralStation** (Subject) which would update the attribute areaObserved, so each robot would only search a given space. However, we have realized that usage of Subject-Observer pattern makes sense only if Subject is communicating with Observers of different classes.

2. Improved idea and new diagram

We have decided to change the way robots are used to explore the environment. We have decided that we will only use one robot at the time, but it will be operated by instances of three different operators (Observers), namely: driver - which will dictate the way our robot moves; photographer - which will use the camera installed on our robot; navigator - which will adjust the path for the driver to follow to reach the object. Our central station will have 5 states so it can effectively distribute the job between the operators (Observers) and itself as well. Below you can find our new class diagram which represents our new setup.



Class Subject:

The class **Subject** implements the Subject-Observer design pattern. It is used so that CentralStation can coordinate the job of the Operators (Observers). All attributes and operations used in the class are derived from the design pattern.

Attributes:

- private Observer[]: observers - list of type Observer which keeps track of all the objects currently in Subject-Observer relation with a specific subject
- private Integer: state - in the case of Subject class, attribute state of type integer is used to keep different values which are then used to coordinate the job of the observers

Operations:

- public Integer getState() - a public method which is used mostly by the Subject's Observers to access the value of a private attribute state
- public void setState(Integer: state) - a public method which is used to manipulate the value of Subject's state attribute. This method also informs all the Subject's Observers currently in the Observer[]: observers list, that the state was changed
- public void attach(Observer: observer) - a public method which is used when a new object of a class Observer want's to set up the relation with a given Subject

- public void notifyAllObservers() - a public method used here in the method setState(). It informs all the Subject's Observers currently in the Observer[]: observers list, that the state was changed

Associations:

The class **Subject** is a superclass for the CentralStation.

Class Observer:

The class **Observer** implements the Subject-Observer design pattern. It is used so that CentralStation (Subject) can coordinate the job of Driver, Photographer and Navigator (Observers). All attributes and operations used in the class are derived from the design pattern.

Operations:

- public void Update() - a public method used when the Subject which is in relation with a given Observer changes its state. In the update method an object of a class Observer checks if the state after the change corresponds with a certain Observer's behavior

Associations:

The class **CentralStation** inherits from the class Observer.

Class CentralStation extends Subject:

The class **CentralStation** is the most important part of our system. Its 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. It is an implementation of two design patterns namely, the Singleton and the Subject-Observer pattern.

Attributes:

- CentralStation: instance - typical attribute which enables correct implementation of the Singleton design pattern. Pointer to the one and only instance of a class CentralStation
- Integer: boxesToBeFound - integer which keeps the number of boxes that is still to be found in the environment. Vital for the mission progress assessment (boxesToBeFound == 0 -> Mission Completed)
- Point3d []: listOfBoxes - list of a java specific type Point3d which keeps track of the coordinates of the boxes already found by the robot. It is a vital source of information not to take the same box in the environment as two separate ones. Whenever a message of a box being found comes to the CentalStation, the coordinates of a robot are cross-checked with the list to find out whether the box was already found or not.
- Integer: numberOfBots - simple integer which keeps track of the numbers of Robots exploring the environment

Operations:

- public CentralStation: getInstance() - typical operation which enables correct implementation of the Singleton design pattern. It returns reference to the one and only instance of a given Singleton class
- public void decreaseBoxesToBeFound() -operation which decreases the value of boxesToBeFound attribute by one every time the Photographer informs that the object detected is a **Box** and it wasn't detected before (is not nearby any point kept in the listOfBoxes attribute)
- @Override public void setState(Integer: State) - overriden Subject class operation. The most important operation in the class CentralStation. Allows the communication between Subject and the Observers. It is also responsible for keeping track of the progress of the mission. If the number of boxesToBeFound is decreased to zero it informs user that the mission is completed, prints all coordinates of the found boxes and changes state to 6
- public void addRobot() - method which creates an instances of Driver, Photographer and Navigator and attach them to the list of observers. Then it creates an instance of a robot and adds it to the environment. At the end it increases the number of robots.

Associations:

CentalStation is associated with **Environment** class as it is responsible for adding the instances of a class **Robot** to the environment. It also inherits form Subject.

Class Driver extends Observer:

The class **Driver** implements the Subject part of the Subject-Observer pattern design. It's main purpose is to operate robot actuators so it can move freely around the environment. It's actions correspond to CentralStation's states 2 and 5. State 2 is an initial state. It also indicates that Photographer couldn't find any pictures with boxes in it. For state 2, driver makes the robot go randomly around the environment. When the hit is detected it set the CentralStation state to 3. State 5 indicates that Photographer has found the best picture with a box in it, Navigator has computed the best way to get to the box and driver makes the robot follow a straight line to it.

Attributes:

- Double: angle - the value of angle determines the value of y-axis rotation of the robot. The value of angle is set by the Navigator. After it is set driver rotates the robot and makes it follow the straight line until it reach the box.

Operations:

- public void assignRobot() - method used to assign an instance of a class Robot to a Driver so he can operate it.
- public void driveToGoal() - method used to rotate the robot and make it follow a straight line until it reaches the Box
- public void setAngle(Double: angle) - public class which allows navigator set the angle of Robot rotation.

- public void hitDetected() **-** if the method is executed driveMode is set to 1 and setState(6) is called

Associations:

The class **Driver** is associated with a single instance of Class robot. This association can be described as “Driver OPERATES Robot”. It is also associated with CentralStation implementing the Subject-Observer design pattern. Driver’s actions are based on the state of the CentralStation which he is assigned to.

Class Photographer extends Observer:

The class **Photographer** implements the Subject part of the Subject-Observer pattern design. It’s actions correspond to CentralStation’s state 3. State 3 is set by a Driver in the event of hitting an obstacle. For state 3 Photographer’s main role is to take pictures of the Robot’s surroundings and to analyse them later in search for the picture with the highest ratio of yellow over the other colors. If such photo is detected, it’s number is assigned to bestPhoto attribute.

Attributes:

- integer: bestPhoto - saves the number of photo in which the ratio of yellow color over the rest of the picture is the highest. It is later used by navigator to calculate the angle of robot rotation

Operations:

- private takePhotos() - used to photograph the environment and classify the pictures in search for the one with the highest [yellow/other colors] ratio
- public int getBestPhoto() - simple method which returns the number of the best photo determined by takePhotos()

Associations:

The class **Photographer** is associated with a single instance of Class robot. This association can be described as “Photographer OPERATES Robot”. It is also associated with CentralStation implementing the Subject-Observer design pattern. Photographer’s actions are based on the state of the CentralStation which he is assigned to.

Class Navigator extends Observer:

The class **Navigator** implements the Subject part of the Subject-Observer pattern design. It’s actions correspond to CentralStation’s state 4. State 4 is set by a Photographer in the event of finished photo analysis. It’s main purpose is to calculate and update the angle of rotation of the robot for the driver.

Attributes:

- integer: bestPhoto - saves the number of photo in which yellow color takes the most of its area it is used by navigator to calculate the angle of robot rotation by multiplying it with the angle 360/total number of photos taken

Operations:

It has no methods. It uses Driver's public setAngle() method to update the rotation angle of the robot.

Associations:

The class **Navigator** is associated with a single instance of Class Photographer and single instance of class Driver. Those associations can be described as "Navigator TAKES THE NUMBER OF BEST PHOTO FROM Photographer" and "Navigator SETS ANGLE FOR Driver. It is also associated with CentralStation implementing the Subject-Observer design pattern. Navigator's actions are based on the state of the CentralStation which he is assigned to.

Class Robot extends Agent:

The class **Robot** in our design is really simple. It lacks complex operations as we have decided that it will only act as a tool for its Operators (**Driver, Navigator, Photographer**). It is implemented to deliver the functionality of the sensors, actuators and cameras which are operated by Driver, Navigator and Photographer as stated above. It implements the basic methods extending the **Agent** class provided in simbad library.

Attributes:

- bumperBeltSensor: bumper - attribute which represents the bumper sensor used to detect contact with other objects.
- cameraSensor: camera - attribute which represents the camera sensor which is used to analyse the Robot's surroundings

Operations:

- public void hit() - method inherited from class Agent
- public void initBehavior() - method inherited from class Agent
- public void performBehavior() -method inherited from class Agent. Additionally this method is responsible for rotating the robot and calling the method takePhoto 10 times in order to add the images to the list, which will be used by photographer for picture classification.
- public void move() - method inherited from class Agent
- public BufferedImage takePhoto() – Take photo and return it for further processing purposes

~~Associations:~~

~~**Robot** is associated with CentralStation as the robots are added to the environment as a part of a CentralStation code.~~

Class BoxFactory:

The class **BoxFactory** implements the Factory Method pattern design. It's main purpose is to create and return instances of a class Box during the run-time

Operations:

- Box : getBox(Environment: environment) - used to create and return the instances of the class Box

Class Environment:

The class **Environment** represents the 3D Environment which will be explored by our robots. It implements Singleton design pattern.

Attributes:

- Environment: instance - typical attribute which enables correct implementation of the Singleton design pattern. Pointer to the one and only instance of a class CentralStation

Operations:

- public Environment: getInstance() - typical operation which enables correct implementation of the Singleton design pattern. It returns reference to the one and only instance of a given Singleton class.

Associations:

Class UserInterface:

The class **UserInterface** represents the user interface through which user can declare the number of boxes in the environment and set their position.

Attributes:

- integer: numberOfBoxes - private integer which records the number of boxes user wants to add to the environment

Operations:

- void addBoxes(Environment: environment) - operation which initiates the process of adding boxes by the user. Takes Environment as parameter. In the body it asks user for the number and parameters of the boxes to be found, uses **BoxFactory** getBox method to access new instances of a **Box**.

Associations:

UserInterface is associated with **BoxFactory** and **Environment** classes. BoxFactory association is necessary to access its method getBox, through which **UserInterface** can get multiple instances of the class **Box** at run-time. To add them to the environment, the association between **UserInterface** and **Environment** is vital.

DataType Coordinate:

We are going to use DataType to represent the position of the boxes already found by a robot. It calculates internally the coordinates of the box derived from the position of the robot.

Attributes:

- double: latitudeX - an attribute of type Double representing the latitude of the box found

- double: longitudeZ - an attribute of type Double representing the longitude of the box found

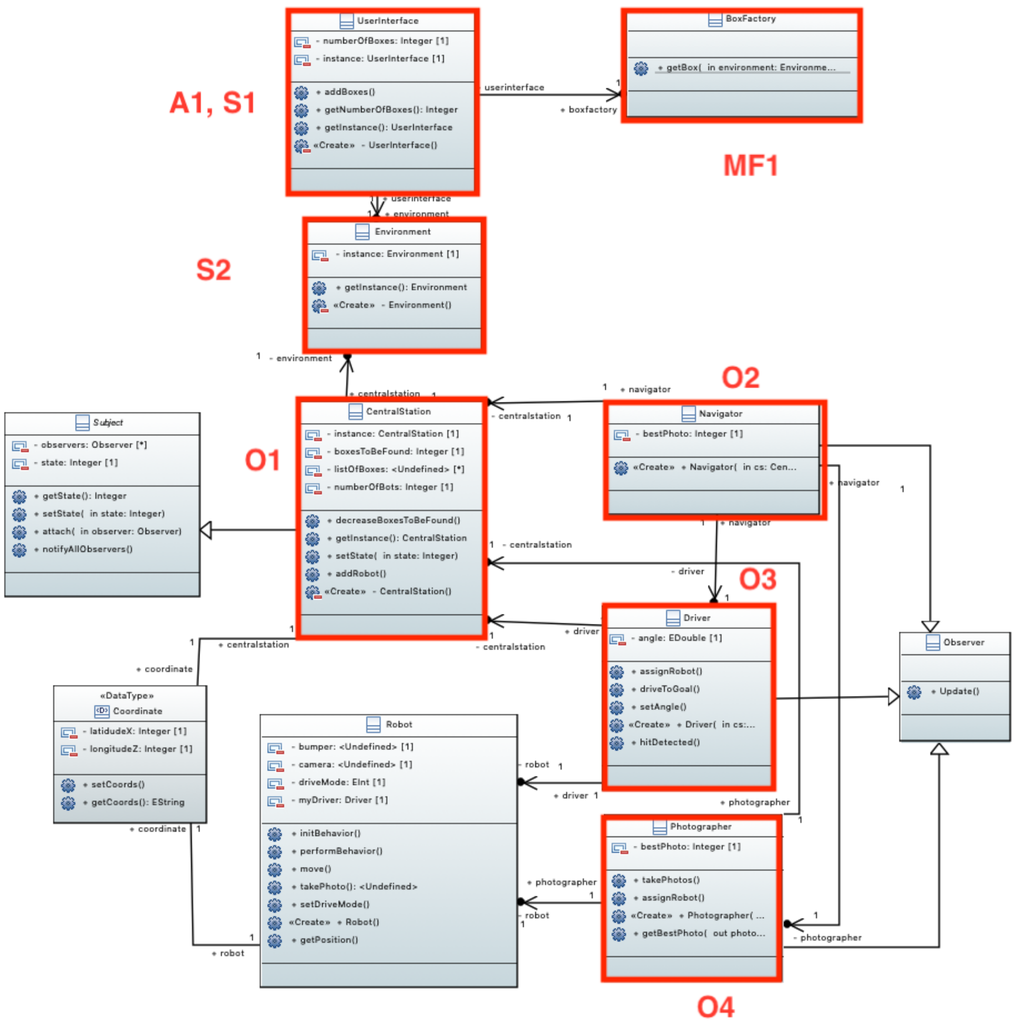
Operations:

- void setCoords(Point3d: point) - method used to set the coordinates described above

Associations:

Coordinate is associated with Robot and CentralStation as it is used in both classes

2. Applied design patterns



ID	S1, S2
Design pattern	Singleton
Problem	There can only be one instance of a class Environment and Centralstation each, at run-time
Solution	We have applied the Singleton design pattern while creating both Environment, UserInterface and CentralStation classes
Intended use	The moment our program starts is creates an instance of Environment, User interface and CentralStation which can be later used and referred to at run-time.
Constraints	-
Additional remarks	-

ID	DP1
Design pattern	Factory Method
Problem	UserInterface is not able to actively generate the desired number of instances of a class Box to be added to the environment at run-time
Solution	We have designed and implemented BoxFactory class which enables creating the instances of a class Box actively at run-time
Intended use	UserInterface allows user to choose the number of the

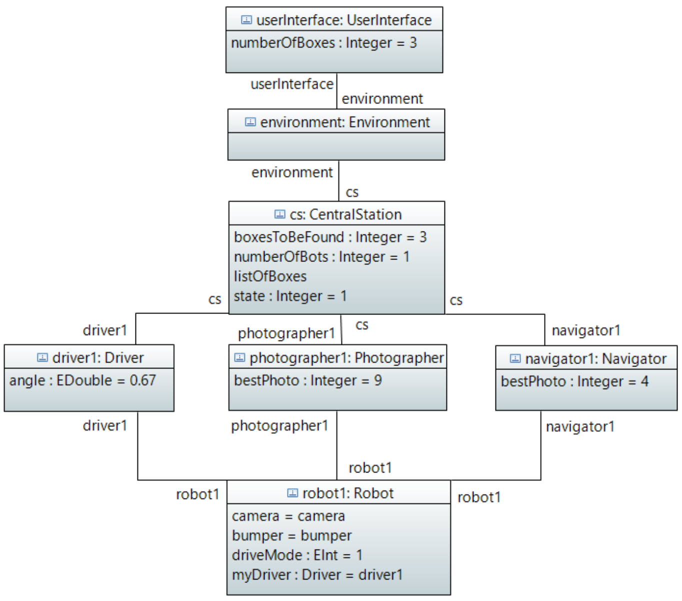
	boxes to be added to the environment together with their coordinates. It uses the method getBox() provided by BoxFactory class to generate ready-to-add instances of class Box which are later added to the environment.
Constraints	-
Additional remarks	-

ID	A1
Design pattern	Adapter (WRAPPER)
Problem	We need an interface through which user can declare the number of boxes and their coordinates in the environment without manipulating the environment code itself
Solution	We have implemented the UserInteface class following the Adapter design pattern. Environment is it's legacy component. It allows to add objects to it without hardcoding them in the Environment code before running the code
Intended use	EnvironmentWrapped will act as a form of a user interface which will enable user to choose the desired number of boxes to be placed in the environment together with their coordinates.
Constraints	-
Additional remarks	-

ID	O1, O2, O3, O4
Design pattern	Observer
Problem	We need an effective mean of communication between CentralStation, Driver, Navigator and Photographer. Only one of them should be able to use robot at the time.
Solution	We have applied the Subject-Observer design pattern in which CentralStation is a Subject and Driver, Navigator and Photographer are the Observers
Intended use	Everytime robot is created it has assigned one Driver one Photographer and one Navigator. They operate same robot consecutively trying to find a Box. As the state of CentralStation changes, all observers recieve an update which gives them the possibility to use the robot. It is very helpful as we have only CantralStation that dictates the state of each Observer whether it should operate the Robot or nor
Constraints	-
Additional remarks	-

3. Object diagram

Author(s): Jiyoung Oh, Karol Komorniczaka



This is a snapshot of our system while Photographer is using a Robot to take pictures and two boxes has already been found.

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, all Boxes are added to the environment and their number is saved. Then an instance of CentralStation is instantiated. After that we and change the state of CentralStation to 0. In state 0 CentralStation checks the number of boxes and adds robots to the environment using the addRobot() method which also creates an instance of Driver, Observer and Navigator each and assign them to a single Robot. State is changed to 1. Now central station checks the mission progress and checks the number of boxes still to be found. If it’s bigger than 0 it changes state to 2 and updates all observers. Update with state 2 informs the Driver that it’s his turn to start moving the robot. When it hits something it sets state to 3. At state 3 Photographer starts taking the pictures and analyse them. If he finds a photo with yellow color in it it saves it’s number. If the ratio of color yellow over other colors is higher than 0.45 it indicates to central station that the box was found. NumberOfBoxesToBeFound is decreased and coordinates of the box are added to the list and state is set to 1. 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 and changes state to 4. If he didn’t find a photo he sets state back to 2. At state 4 navigator sets new angle for the driver so he knows where to go. The process is repeated until all boxes are found.

4. Code generation remarks

Author(s): <Maciej Juzon>

In order to generate the code, we used 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 by right-clicking on the root element in the model explorer. This created a separate folder with all our classes and interface in the workspace. However, the generated code in the beginning was incompatible with the simbad framework.

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 one.

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).

5. Implementation remarks

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

After finishing all the tasks mentioned above we were able to start implementing our system. Because, we decided to abandon the idea from our first deliverable, we had to basically implement everything from the scratch. However, thanks to the fact that the generated code was really clear, the whole process was not as time consuming as we thought. For the first time we saw the advantage of designing the software properly before coding, as previously we always preferred the approach: Do something and let see what is going to happen. When we started working on deliverable 2, we went straight to coding and we thought that we can design a proper diagram based on that, but it turned out to be a complete mess and thus we had to start over.

Thanks to proper plan and a decent class diagram we knew exactly what to do and what we want to achieve. Generated code allowed us to split the work fairly between two people and made the process of putting everything together really easy. However, implementing our system resulted in some minor changes in our class diagram, because some attributes which we firstly wanted to be private, actually needed to be public. Beside that we also come up with some new methods that were not included in the beginning in our class diagram, so we had to add them.

For now, we have implemented all of the basic functionalities of our system, so one robot can drive around the environment and can be controlled by our observers, namely: Photographer, Navigator and Driver, each one of them has different functionality and depending on state of our system they take control over the robot and make it perform certain tasks. All the observers are controlled by central station, which is responsible for monitoring the progress of the mission and notifying the observers when state changes. However, we still need to make some improvements and fully implement our data type which is the exact position of the found box. Our photographer is responsible for making the robot take the photo and then classifying it based on the amount of yellow pixels visible on it. Currently we only have a demo of our working classifier and we need to improve it to achieve greater accuracy. Also for deliverable 3 we will try to provide a system with more than just one robot driving around the environment, as our central station is designed in such a way, that it can support more than one robot.

6. References

-