Software design
Team project – Deliverable 2

Team number: 4

Team members:

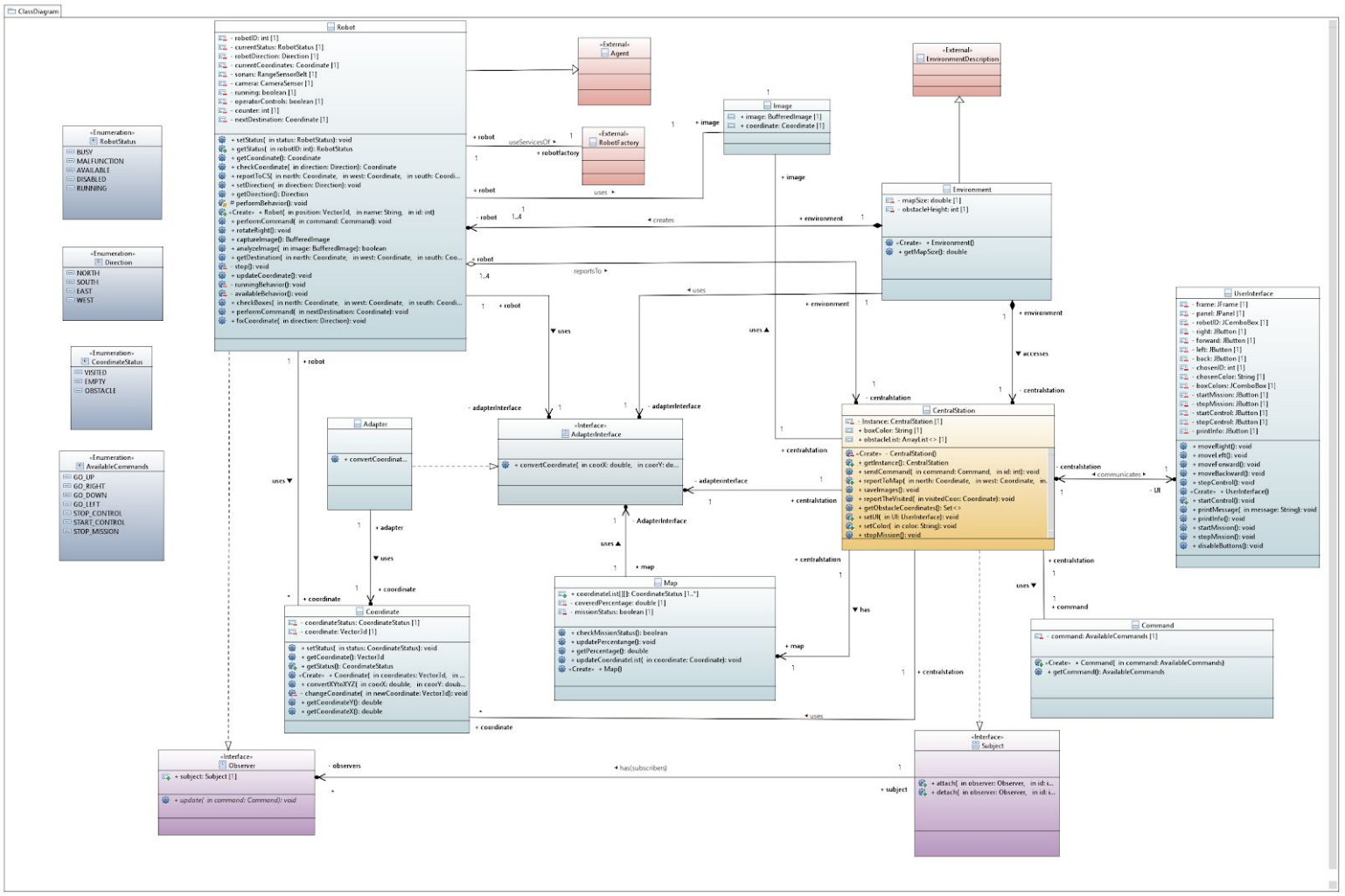
Name	**Student Nr.**	**E-mail**
Basel Sammor	2608511	b.s.a.sammor@student.vu.nl
Yasin Aydın	2657725	yasinaydin@sabanciuniv.edu
Connor de Bont	2599228	connordebont@gmail.com
Faruk Simsekli	2657316	faruksimsekli.7@gmail.com

Table of Contents

1. Class Diagram

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor De Bont.

This chapter contains the specification of the UML class diagram of our system, together with a textual description of all its elements. Some of the external classes used that were not important to show were added in a different diagram just to make the class diagram less cluttered.



Class Robot

The **Robot** class represents *robot* objects that are created in the environment to complete tasks that are given to them. Tasks are given to accomplish the general mission in the system, which is to find the specified boxes. These tasks can be only given by Central Station or Operator with *command* objects. *Robot* objects collect information from the unknown map and transfer them to the *central station* and operator while using the algorithm that avoids obstacles and finds a path for *robots*.

Attributes

- **int robotID:** an integer to identify *robots* from others which shall be used to give different *robots* different tasks.
- **Coordinate currentCoordinate:** the current coordinate of the *robot*.
- ~~Coordinate initCoordinate: the initial coordinate of the robot.~~
- **RangeSensorBelt sonar:** there are 4 sonars placed around each *robot that detect obstacles and calculate distance*.
- ~~RangeSensorBelt bumpers: there are 8 bumpers that detect collisions~~
- **CameraSensor camera:** there is a camera that will be taking images of obstacles.
- **RobotStatus currentStatus:** the current status of the *robot*, which can be ~~four~~ five different states described in the **RobotStatus** enumeration
- **Direction robotDirection:** the current direction that the *robot* is looking at, which can be four different directions from **Direction** enumeration
- **boolean running:** this static boolean acts as a guard and is checked to determine if the robots should continue with the mission or the mission is over.
- **int counter:** an integer to determine when a robot should rotate and analyze images to allow the camera to update it's view.
- **Coordinate nextDestination:** stores the next coordinate the robot has to move to.
- **boolean operatorControls:** a boolean that determines if a robot is in manual control mode or not.

Operations

- **Constructor Robot(Vector3d position, String name, int id)**
- the four possible directions to the central station. It also reports its own coordinate.
- **void setStatus(RobotStatus status):** sets the currentStatus of the *robot* to the value in the parameter
- **RobotStatus getStatus(int robotID):** returns the currentStatus of the *robot*
- **Coordinate getCoordinate():** this function returns currentCoordinate of the *robot*
- **Coordinate checkCoordinate(Direction direction):** this function checks the direction provided as a parameter and returns it's status.
- **void reportToCS(Coordinate north, Coordinate west, Coordinate south, Coordinate east):** Reports the status empty or obstacle (as described in CoordinateStatus enumaration) of the four possible directions to the central station. It also reports its own coordinate.
- **void setDirection(Direction direction):** this function shall turn the *robot* to the direction of input parameter and update the current robotDirection.
- **Direction getDirection():** this function returns the current direction of the *robot* according to the **Direction** enumeration
- **void performBehavior():** ~~the overall algorithm in the system is coded inside this function, which makes sure that robot avoids obstacles and in the meantime keeps~~

~~sending information to the central station. Also, before sending reports its checks near obstacles and calls analyzeImage() to check if it is the colored box that it is looking for.~~ this function is called continuously by the simulator and is used to decide what behavior the robot should follow according to its currentStatus. Depending on the status, a different function is called repeatedly.

- **void performCommand(Command command):** the *robot* shall end it's a current task and starts to perform the requested command.
- **void performCommand(Coordinate nextDestination):** overridden performCommand where the robot moves to the provided nextDestination.
- **void rotateRight():** this function rotates *robot* ninety degrees to the right while keeping *robot* steady
- **BufferedImage captureImage():** this function will capture the robot's camera point of view and returns it as a BufferedImage for further analysis.
- **boolean analyzeImage(BufferedImage image):** this function will analyze the returned image from **captureImage()** for the specified mission color.
- **Coordinate getDestination(Coordinate north, Coordinate west, Coordinate south, Coordinate east):** this function checks the four directions around the robot and sets the nextDestination as one of those directions if it's empty. In some cases where are all visited, a random coordinate is chosen.
- **void stop():** this function sets the robot status as disabled and stops everything to end the mission.
- **void updateCoordinate():** updates the currentCoordinates of the robot by using the **getCoords()** function from the superclass.
- **void runningBehavior():** this function consists of part of our algorithm where it only executes after a destination is set and the robot is moving or "RUNNING" towards it until it reaches the destination.
- **void availableBehavior():** this function consists of part of our algorithm where it executes when the robot is AVAILABLE and makes up the other part of our algorithm which takes care of image capturing, reporting information, and choosing destinations.
- **void CheckBoxes():** this function takes care of capturing images and analyzing the boxes around the robot to determine if they fit the chosen color.
- **void fixCoordinate():** this function is used in order to fix the robot's coordinate due to the inaccuracy of double values when used as the coordinate's X and Y values.

Associations

The **Robot** class is associated with the **Central Station** class. Robots shall contain a reference to the central station so *the central station* is a part of *robot*, which expresses a weak belonging between them(shared aggregation). The reason behind associations is that robots shall use **reportToMap()** and **reportTheVisited()** function to update *map* and access some of its attributes. One to four *robots* can be created and those can only have one *central station* because of the singleton design.

The **Robot** class is associated with the **AdapterInterface** class. The association is from **Robot** to **AdapterInterface**, since *robots* act as clients and use interface to convert coordinates in the way we implemented it.

The **Robot** class is associated with the **Coordinate** class**.** One robot can have many coordinates at the same time. *Robots* shall use *coordinates* when reporting to the *central station*. The navigability is undefined since in the code generation we faced issues with the creation of a coordinate variable.

The **Robot** class is associated with the **Image** class**.** One robot can have many Images at the same time. *Robots* shall use *Images* when capturing photos of boxes and updating to the *central station's list*. The navigability is undefined since in the code generation we faced issues with the creation of a map variable.

Class CentralStation

The **CentralStation** class represents a place where we keep track of the *map* and the status of its coordinates. It will contain a map that represents each point in the *environment* as a coordinate with its respective status. Another objective is to control and send commands to *robots* ~~in order to make the *robots* move efficiently~~. Additionally, manual control will be done by communicating through the *central stations* to the *robots*. It also holds the images of the boxes that match the chosen color.

Attributes

- **CentralStation Instance:** this attribute instantiates a private static object of CentralStation.
- **String boxColor:** this attribute contains the chosen box color that the operator chooses.
- **ArrayList<Image> obstacleList:** this arraylist will hold the images captured by robots of the boxes that match the box color .

Operations

- **Constructor CentralStation():** this function is the constructor for CentralStation and it is private to make sure that only a single object of CentralStation can be instantiated.
- **CentralStation getInstance():** this function shall return the one and only one object of CentralStation available. this function returns the *instance* object
- **void sendCommand(Command command , int robotID):** this function shall send specific tasks to be executed by the mapping robots. this function shall send the task to the *robot* which has the same robotID as requested.
- **void reportToMap(Coordinate north, Coordinate west, Coordinate south, Coordinate east):** This will report the coordinates given by the *robot's* reportToCS() operation and their status to the map.

- **void saveImages():** this function shall be called at the end to save the images stored in the list on disk for later.
- **Set<String> getObstacleCoordinates():** this function returns a set containing the coordinates of the coordinates where the boxes were found.
- **void setUI(UserInterface UI):** sets the UserInterface attribute to the provided parameter.
- **void setColor():** sets the boxColor attribute to the provided parameter.
- **void stopMission():** this function performs the required tasks to end the mission and informing the robots to stop.
- **void reportTheVisited(Coordinate visited):** This will report the coordinates given by the *robot* and their status to the map.

Associations

The **CentralStation** class is associated with the **Coordinate** class**.** One central station can have many coordinates at the same time. *Central station* shall use *coordinates* when reporting to the *map*. The navigability is undefined since in the code generation we faced issues with the creation of a coordinate variable.

The **CentralStation** class is associated with the **Map** class**.** The *central station* will report to *map* to update the coordinates and change when necessary. Also, *map* is holding the coveredPercentage value so *central station* shall check it with checkMissionStatus() to see if game has to be stopped.

The **CentralStation** is associated with the **Image** class. It will be using it to create an arraylist of Images to store the objects. The navigability is undefined since in the code generation we faced issues with the creation of a coordinate variable.

The **CentralStation** class is associated with the **AdapterInterface** class. The association is from **CentralStation** to **AdapterInterface**, since *central station* act as clients and use interface to convert coordinates in the way that is implemented.

The **CentralStation** class is associated with the **Command** class. The association is from **CentralStation** to **Command.** A *command* object shall be created if the user want to manually control a particular *robot*. So, the *central station* should be capable of creating *commands* and using them to send tasks to the related *robot*.

The **CentralStation** is associated with the **UserInterface** class. It needs to be able to communicate with the operator through the UserInterface.

Class UserInterface

The **UserInterface** class represents the operator's *interface* that allows the operator to manually control *robots.* Also from this *interface*, the operator can extract information that a *robot* holds.

Attributes

- **JFrame frame:** this attribute will represent the frame for the Manual Control GUI.the window that
- **JPanel panel** this attribute will represent a panel for the Manual Control GUI.
- **JComboBox robotID** this attribute will list the ID of available robots that can receive manual commands from the operator.
- **JComboBox boxColors:** this attribute will list the available colors that can be chosen by the operator for the mission goal.
- **JButton stopMission:** this attribute will represent the go stop mission button in the manual control GUI
- **JButton startMission:** this attribute will represent the go start mission button in the manual control GUI.
- **JButton startControl:** this attribute will represent the go start control button in the manual control GUI.
- **JButton stopControl:** this attribute will represent the go stop control button in the manual control GUI.
- **Int chosenID:** this integer will represent the chosen ID from the robot in the robotID combobox.
- **String chosenColor:** this string will represent the chosen color from the robot in the boxColors combobox.
- **JButton printInfo:** this attribute will represent the retrieve Info button in the manual control GUI.
- **JButton Forward** this attribute will represent the go forward button in the Manual Control GUI.: The operator commands the *robot* to go forward
- **JButton Right** this attribute will represent the go right button in the Manual Control GUI.: The operator commands the robot to go right
- **JButton Back** this attribute will represent the go back button in the Manual Control GUI.: The operator commands the robot to go back
- **JButton Left** this attribute will represent the go left button in the Manual Control GUI.: The operator commands the robot to go left

Operations

- **Constructor UserInterface()**
- **void moveRight():** this function will stop autonomous movement and will manually command a specific robot to move right.
- **void moveLeft():** this function will stop autonomous movement and will manually command a specific robot to move left.

- **void moveForward():** this function will stop autonomous movement and will manually command a specific robot to move forward.
- **void moveBackward():** this function will stop autonomous movement and will manually command a specific robot to move backward.
- **void stopControl():** this function will cause the manually controlled robot to start autonomously moving around the environment again.
- **void startControl():** this function will enable manual control of the chosen robot ID.
- **void startMission():** this function will start the mission, a color must be chosen before this is allowed.
- **void stopMission():** this function will stop the mission prematurely by the operator.
- **void printInfo():** this function will print the info of the boxes as a message in the GUI.
- **void printMessage(String message):** this function will print the provided parameter to the operator.
- **disableButtons():** this function will disable all of the buttons in the GUI except the retrieve info when the mission ends to signal nothing can be initiated.

Associations

The **UserInterface** class is associated with the **CentralStation** class**.** Association is from **UserInterface** to **CentralStation**, since the operator will manually control *robots* with the **sendCommand()** function.

Class Environment

The **Environment** class represents and creates the current *environment* that the system is using, which includes *robots*, *walls*, and *boxes*.

Attributes

- **double mapSize:** it is a double value that holds the size of the map
- **Int obstacleHeight:** a value that is the height of the obstacles in the environment

Operations

- **Constructor Environment()**
- **void getMapSize():** returns the size of the map

Associations

The **Environment** class is associated with the **Robot** class. One to four robots will be created in the environment.There is strong belonging between **Environment** and **Robot** since *robots* can not exist without environment. The operations of **Robot** will be accessible from the **Environment** class via *robots* that are created.

The **Environment** class is associated with the **AdapterInterface** class. The association is from **Environment** to **AdapterInterface**, since *environment* act as clients and use interface to convert coordinates in the way that is implemented.

The **Environment** class is associated with the **CentralStation**. This is necessary because *central station* shall be created in environment with *robots*. Also, there is strong belonging between these classes since *central station* can not exist without environment.

Class Map

The **Map** class represents an abstract representation of the *environment* as seen by the *central station*. It holds the status of *coordinates* in the *environment* that are known by the *central station*.

Attributes

- **CoordinateStatus[][] coordinateList:** a 2D array that contains the status of that specific coordinate, the array, and map origin is synchronized in a way that both have (0,0) at the top left corner.
- **double coveredPercentage:** the value that is the area percentage covered by the robots.
- **boolean missionStatus:** it represents whether the mission is over or not

Operations

- **Constructor Map()**
- **boolean checkMissionStatus():** true if and only if 70% of the *map* coordinates are visited.
- **void updatePercantage():** this function updates the percentage of the map that is visited after each movement.
- **double getPercentage():** returns the coveredPercentage of the map.
- **void updateCoordinateList(Coordinate coordinate):** shall update the map coordinates with their statuses as reported by *central station's* reportToMap() and reportTheVisited() operation in the Coordinate array.

Associations

The **Map** class is associated with the **AdapterInterface** class. The association is from **Map** to **AdapterInterface**, since *map* act as clients and use interface to convert coordinates in the way that is implemented.

~~The **Map** class is associated with the **Environment** class. It is one to one relation since there is will be only one map and one environment. The reason for this association is that **Map** must have access for mapSize, which is in **Environment** class, in order to create the same size array~~

~~to store coordinates inside of it.~~

Interface AdapterInterface

The **AdapterInferface** interface represents the expected interface for the client class. It is the class that the **Adapter** implements. The adapter shall implement its operations.

Operations

- **Vector3d convertCoordinate(double coorX, double coorY):** the function that is called by clients of adapter, which has implementation and explanation in Adapter class.

Class Adapter

The **Adapter** class represents link between the incompatible Client and Adaptee classes. The adapter implements the AdapterInterface and contains a private instance of the Adaptee class. When the client executes **convertCoordinate** function on the AdapterInterface, **convertCoordinate** function in the adapter translates this request to a call to **convertXYtoXYZ** function on the internal Adaptee instance.

Operations

- **Vector3d convertCoordinate(double coorX, double coorY):** this function returns vector3d *coordinate* by using **convertXYtoXYZ()** from (Adaptee) **Coordinate** class.

Associations

The **Adapter** class is associated with the **Coordinate** class. The association is from **Adapter** to **Coordinate**. The reason behind this association is the adapter design pattern. **Coordinate** class is adaptee and **convertCoordinate()** function must use the operations inside of **Coordinate.**

Class Coordinate

The **Coordinate** class represents each 3-D coordinates in the *map*. These *coordinates* shall be converted into X- and Y- values with the *adapter*.

Attributes

- **Vector3d coordinate:** represents the xyz value of a coordinate in the map.
- **CoordinateStatus coordinateStatus:** represents the status of each coordinate in the map.

Operations

- **Constructor Coordinate(Vector3d coordinate, CoordinateStatus status)**
- **Void setStatus(CoordinateStatus status):** this function sets the status as specified in the input parameter.
- **Vector3d getCoordinate():** this function returns the xyz value of the coordinate.
- **double getCoordinateX():** returns the X value of the coordinate.
- **double getCoordinateY():** returns the Y value of the coordinate.
- **void changeCoordinate(Vector3d newCoordinate):** updates the coordinate of the object with the provided parameter.
- **CoordinateStatus getStatus():** this function returns the status of the coordinate
- **Vector3d convertXYtoXYZ(double X, double Y):** this function is for adapter pattern, which converts X- and Y- coordinate to *vector3d* object.

Class Command

The **Command** class represents the tool that *central station* shall use to pass tasks to *robots*.

Attributes

- **AvailableCommands command:** holds a command that can be used as a request by other classes.

Operations

- **Constructor Command(AvailableCommands command)**
- **AvailableCommands getCommand():** returns the command attribute.

Class Image

The **Image** class represents a simple data structure which holds a picture and the location where it was taken in the area.

Attributes

- **BufferedImage image:** this holds the image of the box that a robot takes only when it fits the chosen box color.
- **Coordinate coordinate:** this holds the coordinates of where a box's picture was taken to determine its location.

Interface Subject

The **Subject** interface represents an interface that is realized by **CentralStation** class. All the *robots* in the environment that are subscribed to subject shall be notified when the *map* is updated.

Operations

- **void attach(Observer observer, int id):** subscribes a *robot* observer to be notified and updated automatically
- **void detach(Observer observer, int id):** unsubscribes a *robot* observer so as not to be notified and updated automatically
- ~~**void notifyAll():** notify and update all observers~~

Associations

The **Subject** interface is associated with the **Observer** interface. There is a one to many relation between them since many robots can subscribe to one subject in our system. The association is from **Subject** to **Observer** because **Subject** should be able to access **Observer** interface attributes and operations but not the other way around.

Enumeration RobotStatus

The RobotStatus enumeration is composed of four statuses that a *robot* object can have.

- **AVAILABLE:** All components of the *robot* are operational and the *robot* is ready to start any task handed to it. After completing tasks *robot*'s status shall be automatically turned in to AVAILABLE-status.
- **BUSY:** The *robot* is working on given tasks and bust at the moment.
- **MALFUNCTION:** The *robot* has some malfunctioning components, which are stopping the *robot* from starting a new task and follow the provided algorithm. In the case of malfunctioning *robot* shall send a report to the *central station* and operator.
- **DISABLED:** Only *central station* or operator can change the status of the *robot* to DISABLED-status. In this state, the *robot* cannot perform any function.
- **RUNNING:** the robot is currently moving to the new destination.

Enumeration Direction

The Direction enumeration is composed of four available directions.

- **NORTH:** The direction that the (0,-1) vector is directed to.
- **EAST:** The direction that the (1,0) vector is directed to.
- **SOUTH:** The direction that the (0,1) vector is directed to.
- **WEST:** The direction that the (-1,0) vector is directed to.

Enumeration CoordinateStatus

The CoordinateStatus enumeration is composed of three statuses that a coordinate can have.

- **EMPTY:** The initial state of every coordinate on the map. As soon as, *robots* detect an *obstacle* at that coordinate or they pass through it this state shall change.
- **OBSTACLE:** The state that coordinate turns to when an *obstacle* is detected with the *sensors* of any *robot* that is at most one and a half meter away.
- **VISITED:** The state that coordinate has after a robot passes through it. This indicates that the coordinate is visited so the algorithm shall use this state to determine a better path for *robots* to follow.

Enumeration AvailableCommands

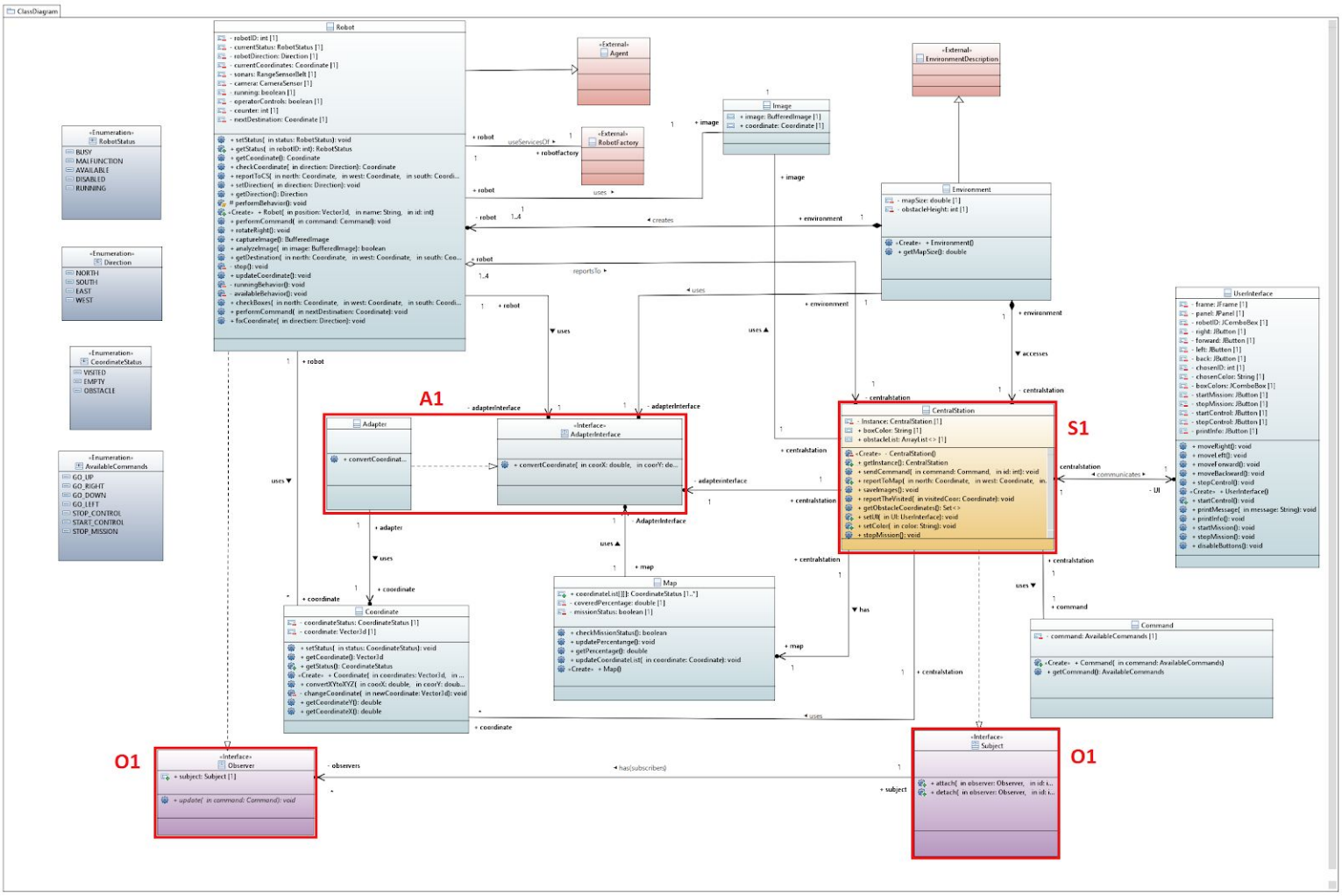
The AvailableCommands enumeration is composed of tasks that can be used to control robots manually. This commands are sent with command objects through central station. Also, the operator shall control robots with these commands.

- ~~**CAPTURE_IMAGE:** the robot captures a bufferedImage and saves it to the specified location~~
- ~~**REPORT_COORDINATE:** the robot reports its current coordinate~~
- ~~**GO_DESTINATION:** the robot goes to the coordinates that are sent from central station~~
- ~~**GO_FORWARD:** the robot goes forward 1 meter from its current position~~
- ~~**GO_RIGHT:** the robot goes right 1 meter from its current position~~
- ~~**GO_LEFT:** the robot goes left 1 meter from its current position~~
- ~~**GO_BACK:** the robot goes back 1 meter from its current position~~
- **GO_UP:** the *robot* goes forward from its current position
- **GO_RIGHT:** the *robot* goes right from its current position
- **GO_LEFT:** the *robot* goes left from its current position
- **GO_DOWN:** the *robot* goes back from its current position
- **STOP_CONTROL:** informs the robot that manual control stopped and to start autonomously moving again.
- **START_CONTROL:** informs the robot that manual control started.
- **STOP_MISSION:** informs the robot that mission has come to an end and to stop functioning.

2. Applied design patterns

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor De Bont.

The following figure represents the UML class diagram in which all the applied design patterns are highlighted graphically.



ID	S1
Design pattern	Singleton
Problem	There must be always one and only one *central station* that communicates with all the *robots* that contain information about the environment during the mission.
Solution	The **CentralStation** class is a Singleton to make sure there is just one at run time that contains the overall information about the environment and the mission.
Intended use	The **CentralStation** class contains a static instance so that all robots can access one and the same anytime and to

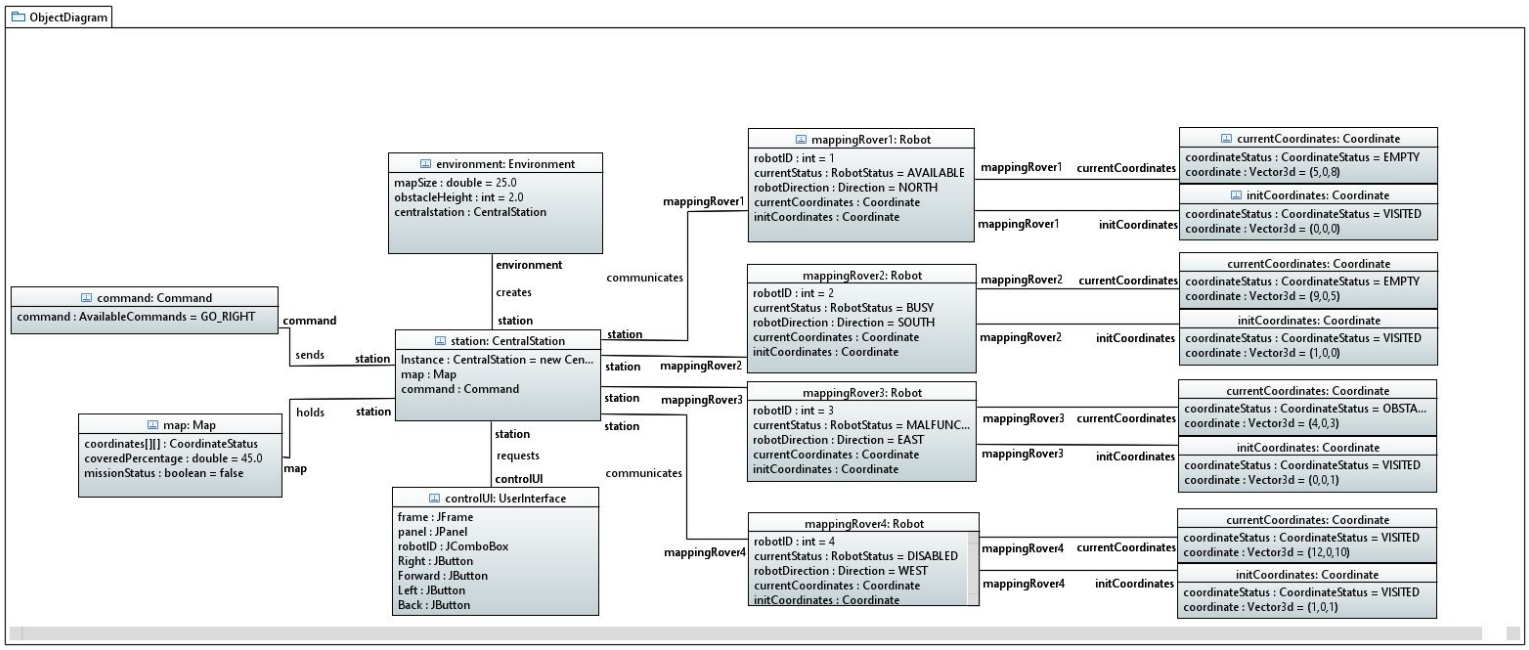
	keep in touch with a constant *central statio*n throughout the mission.
Constraints	The *central station* must always have a direct reference to the robots.
Additional remarks	-

ID	O1
Design pattern	Observer
Problem	The *central station* has to update the *map* with the report that it sends to the **Map** class. The changes that have been made shall be seen by for each and every *robot*.
Solution	Every *robot* in the system is subscribed to the *subject*, *central station* so that whenever a report is sent to the *central station* from a *robot*, all the *robots* shall be notified.
Intended use	To keep the system updated after every event, the changes that every *robot* made shall be known by every other *robot.* In order to keep the system automatically updated **Observer** class shall be used.
Constraints	-
Additional remarks	In the case of a robot is not subscribed to the subject, the *robot* shall be removed.

ID	A1
Design pattern	Adapter
Problem	The Simbad simulator had the origin at the middle of the map, which made our job harder to direct *robots* and create *obstacles* at positions that we wanted. Also, dealing with height parameter every time was unnecessary.
Solution	The map uses 2-D array to store coordinates, which has coordinates[0][0] at the top left corner. So, we will convert (x,y) in a way that origin is at the top left as well. Also **convertXYtoXYZ()** function will be used to convert (x,y) to (x,y,z) in the way that it has origin at the top left.
Intended use	To convert the (x,y,z) 3-D coordinate into (x,y), so that it is easier to use coordinate system. Changing the origin of the coordinate map top make it easier to visualize the map and store it in 2-D array without extra functions.
Constraints	-
Additional remarks	-

3. Object diagram

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor De Bont.



The object diagram is a representation of a snapshot of the system at a specific moment in time.

The CentralStation was instantiated by the Environment and it received a manual GO_RIGHT request from the ControlUI provided to the operator which it will be sending to the appropriate rover.

Four rovers were created, mappingRover1, mappingRover2, mappingRover3, and mappingRover4 and are currently moving around the environment detecting obstacles if their sonar sensors detect any around their 4 directions. mappingRover1 is AVAILABLE for any requests and is currently heading NORTH on an empty coordinate. mappingRover2 is BUSY and is currently executing manual request passed from the central station until manual control is done with. mappingRover3 is currently MALFUNCTION and is not moving but could still be controlled manually to safety if the fault wasn't too severe. mappingRover4 has suffered an irreversible damage and had to be DISABLED.

The rovers receive requests and communicate with the CentralStation during the entire mission to avoid collisions, capture images when detecting obstacles, and moving in an efficient way by not visiting a coordinate twice.

Some objects such as commands and coordinates which are used by more than one class have been not shown repeatedly as it would clutter the diagram.

4. Code generation remarks

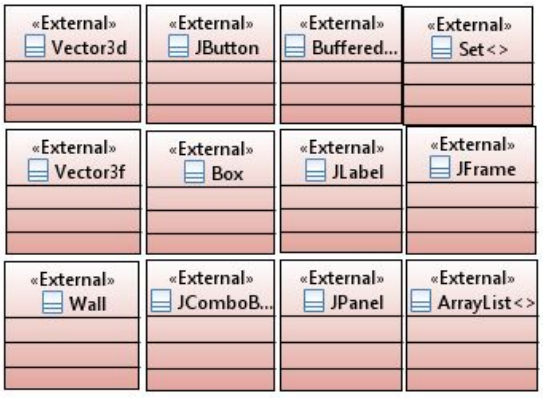
Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor De Bont.

The Java code generated by papyrus from the class diagram went quite well even though we faced some issues. Prior to generating the code we added the JAR files supplied to us in the project skeleton to avoid any errors that could appear from missing them.

The classes had correct attributes and operations for the most part which were generated from attributes that we clearly defined or those declared from associations in the diagram. One of the issues we faced was that even though we included the appropriate return parameters in the class diagram, the generated code did not have return statements in the methods. We were able to manually fix those few errors in the generated code with the help of the error tool by filling them with placeholder statements.

An additional problem we encountered had to do with external types which didn't have their implementation in our class diagram. At first we created empty types with the appropriate names to represent those types. This caused an issue when we generated the code because the import statements that would allow us to use those types were not correct.

One example is the Vector3d class that is available through **javax.vecmath.Vector3d.** We managed to correct the issue by instead applying the external stereotype to the external types and added the suitable import statements to them. This made it so that when we generated the code, any classes using those external types would automatically have the import statements for them.



The generated code serves as a perfect skeleton for our system and the classes of the generated code were left empty for now but the classes generated were used in the implementation part of this deliverable.

5. Implementation remarks

Author(s): Yasin Aydın, Basel Sammor, Faruk Simsekli, Connor De Bont.

In the implementation, we used the code generated from papyrus and filled in some of the empty operations and classes with implementations. The functions we managed to implement are listed below.

Implemented Classes:(Interface and enumerations are not included)

1. Adapter.java
2. Command.java
3. Coordinate.java
4. Environment.java
5. Map.java

Not implemented parts:

From **CentralStation.java** class only **notifyall()** and **detach()** are not implemented.

From the **Robot.java** class, these are the only functions that are not implemented.

1. **performbehavior()**
2. **performCommand(Command command)**
3. **analyzeImage(BufferedImage image)**

(---In deliverable 3 everything is implemented---)

6. References

Following references are used to understand and implement design patterns.

https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

https://riptutorial.com/design-patterns/example/32745/adapter--uml---example-situation-

https://sourcemaking.com/design_patterns/adapter

https://www.tutorialspoint.com/design_pattern/observer_pattern.htm