By: Sergio Puertas Pérez and Xabier Xoán Carricoba Muñoa

REPORT DESIGN ASSIGNMENT, EXERCISE 2

The problem asked us to implement a system to manage the alerts of an Aquarium and its tanks.
Our idea was to create an object Tank with some sensors to measure certain parameters (ph, temperature...), some control devices to take action when these parameters must be fixed and personnel subscribed to certain alerts that receive a report with the issues that occurred concerning their subscriptions. The values of the sensors will be changed via setters.

HOW DOES THE CODE WORK?
We will first create a tank of the aquarium with its name and location. At the moment of creation of the sensors, you will take into account some key parameters: the name of the parameter they measure, the tank to which it is assigned, the default and normal value of that certain parameter, and with which it'll start and two more values: “rangeGood”, which is the deviation from the default value that still doesn't ring an alert, and “rangeOrange”, which is the maximum deviation from the default value which produces an orange alert. This way we will declare the bounds to decide whether an alert must be rung or not.

The alert object is composed by the sensor damaged and its state (ORANGE or RED). For example, a declaration of an alert can be done this way:
Alert alert1 = new Alert (sharksTemperatureSensor,State.RED)
We thought this approach could be better than just passing the level numbers as this makes the communication so much easier, concise and reliable (following the design principles that will be explained after).
Moreover, the object alert has the attribute "value", which stores the value of the damaged sensor in the moment of instantiation. This way, the same sensor can have different alerts concerning the same sensor and state if the values are different: sensor1.setvalue(-3) and sensor1.setvalue(-5) (assuming both are red-level values) will create two different alerts. In addition, the assignation of the alerts is polymorphic: you may either assign a task to the personnel or a control device given a previously instantiated alert or given the parameters (sensor and state).
Finally, a report containing all the personnel and all the alerts they were subscribed and rung is printed.

WHICH PRINCPLES DID WE FOLLOW?
As previously mentioned, to manage the alerts (as objects), we tried to free as much as possible the workload of the control devices. This means that we send alerts with the most important and brief data so that it can be handled correctly and fast. For doing that, the sensors, instead of only measuring the parameters, will also be able to decide whether the variation in those levels provoke an alert and, if so, which type of alert will it be. These ideas are based on the "Minimum Knowledge Principle". In reality, the control device and the sensors don't know how the program works or why they send the messages they send, they just do their job. Another principle that follow this idea is the "Unique Responsibility Principle". As we exposed previously, the unique task of control devices is to change the values of the sensors of the alerts assigned to them to

the default values, and the sensors' unique task is to measure and notify the tank if the values aren't right, which will afterwards notify the observers (control devices).
Furthermore, another SOLID principle we used is the "Open-Closed principle", which can be deduced by the fact that we can make as many sensors as we want and as many control devices as we want without having to change anything in the code and, for example, all alerts will be sent to them.
In addition, another SOLID principle present in our code is de "Dependency Inversion principle", as all classes depend upon the interface Observer, implemented in classes Tank and Control Device. The fact that high level classes do not depend on smaller ones rather than in these interfaces make our code less dependent con concrete classes and more reliable.

Another principles that we can deduce from our code are: "Encapsulate what varies", as getters and setters cover many of the attributes un our classes, the "Kiss" principle, as we always tried to keep the program as simple as possible (everyone could understand the control flow of our program) and the "Yagni" principle, as we only implemented the needed operations in our code.

WHICH PATTERN DID WE USE?
Finally, to create the structure of our code and the management of alerts we followed the Observer pattern. This allows to ensure the communication between different classes in the code that, in this case, must be done efficiently. This pattern defines a low-coupled multi-dependency between objects based on notifications that allow them to update and adapt to changes. This pattern fits perfectly with the "Unique responsibility principle", as this communication allows that many different objects with simple tasks each perform complex operations (as it is to handle de alerts of a whole Aquarium).

We can deduce two observation relations from our design, each one with different goals but that collaborate between them:

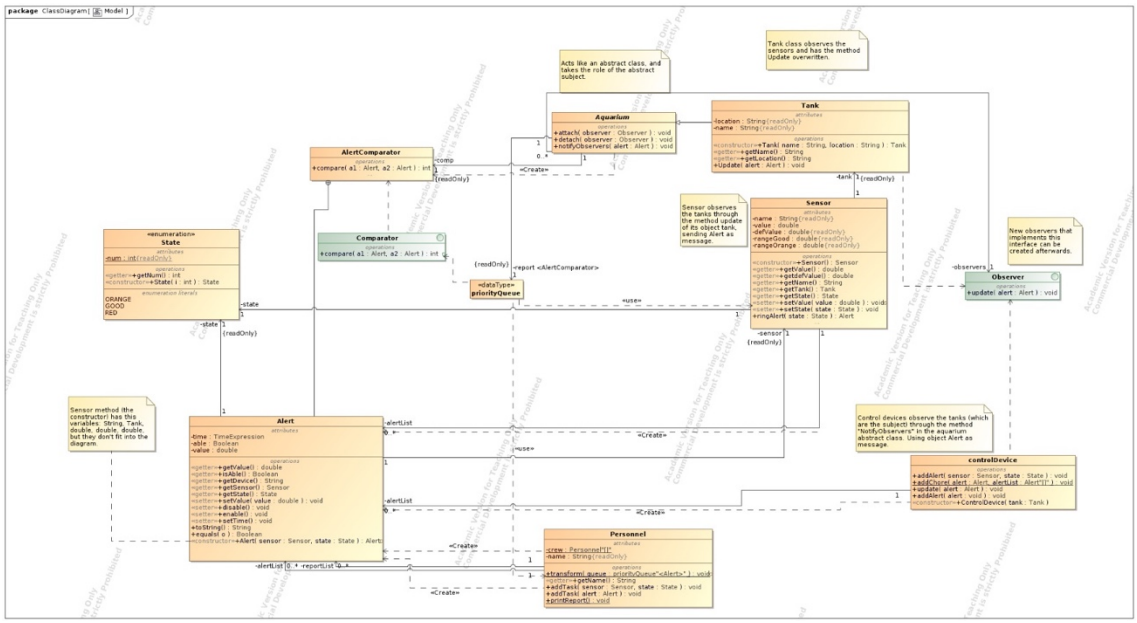
- Firstly, the tanks observe the sensors. They are aware of what is going on inside of them and if the sensors notified anything they could start the protocol to fix the issue: after measuring and creating the alert, the sensor "calls" the tank and sends the alert. This exchange of information is performed using the "Push" technique, as the sensor not only calls the tank but gives them all the information they need to know about the actual case (the alert object itself) so they can handle the situation.

- Secondly, the control devices observe the Tanks (which inheritate the methods of the class Aquarium). This class is the one that deals with the corresponding issues and set the values back to default (that is, the ones they had when they were created). This communication also uses the push method as the tank sends the alert object to the control devices.

By: Sergio Puertas Pérez and Xabier Xoán Carricoba Muñoa

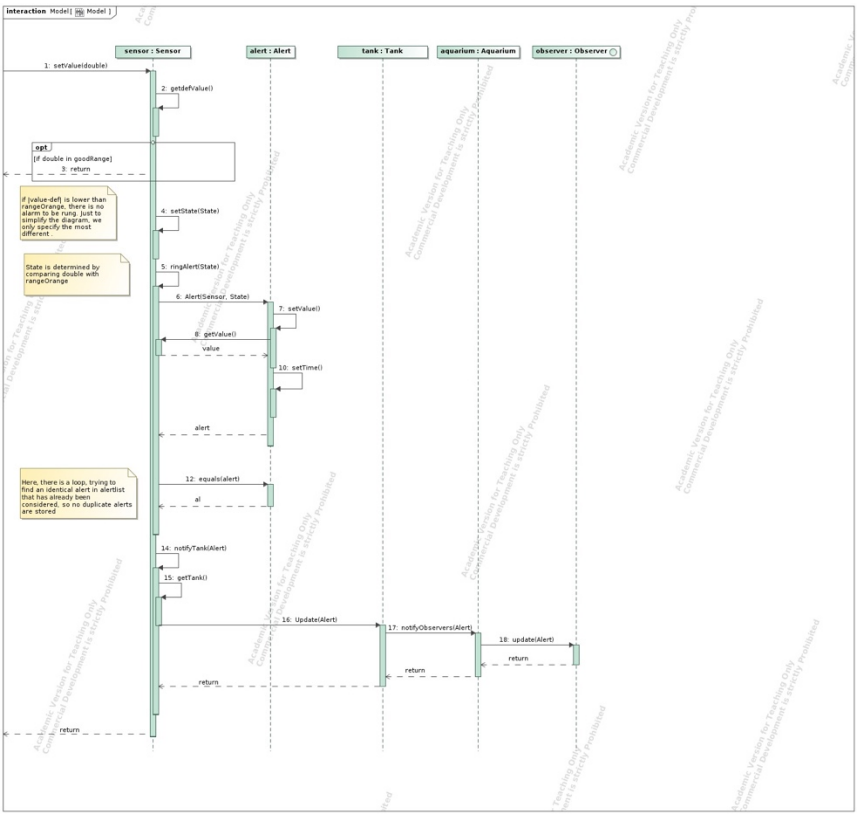
UML DIAGRAMS

CLASS DIAGRAM



As you may notice, it is remarkable to see how many classes are involved in the process and how many relations there are between them, avoiding the concept of "God Class" and pushing for more loosely-coupled designs. Moreover, you can see how the classes Tank and Control Devices implement the interface Observers. There are also notes inside the diagram with this kind of information.

SEQUENCE DIAGRAM



By: Sergio Puertas Pérez and Xabier Xoán Carricoba Muñoa

This sequence diagram corresponds to the method setValue() located in Sensor, which provokes all the changes, measurements and alert in the code. The upper part of the diagram corresponds to the alert creation: if the state changes into ORANGE or RED, a new alert is created. If that identical alert was previously created (for example, when it was assigned to a control device or person), that alert is handled. If not, the new alert is added to the list of alerts and that one is handled.
The lower part shows how the communication between classes works: it's clear how the observers act at the end (thanks to the list of observers who deal and fix the sensors) and how the interface and the design patter is implemented.

In a nutshell, our code performs the demanded exercise in an efficient and, most important, realistic way, which takes benefit of numerous design principles and the observer pattern.