Victor Nathanael Badillo Aldama

Pablo Legide Vidal

E2 REPORT

DESIGN PATTERN FOLLOWED

- **Observer pattern:** the main reason behind the choice to use this pattern was mainly the functioning of sensors in real life and how we imagine they can work at an aquarium tank. As a summary we suppose they measure a parameter changed via the *setParameterLevel* from the **Sensor** class. If this parameter is not in an accepted range for the creatures inside them to be alive (this range is stablished when creating the alarm), it notifies the alarms (concrete observers for each subject) and they are fired which causes them to (which is where they are updated) send the corresponding reports and notify the control devices, in charge of fixing the parameter. Concretely, in this case we are choosing the push model for building this pattern because the different sensors, which are the subject, need to send detailed information about the changes in them to the alerts so when they are updated, sending the reports, they can have all the information necessary such as from which tank they come or which is the level of the parameter which triggers that change in the sensor state, that will notify all its observers. This shows how sensors know which are the needs of the alarms they trigger so they can create a report with the proper information main reason behind the decision of the pull instead of the push model.

DESIGN PRINCIPLES FOLLOWED

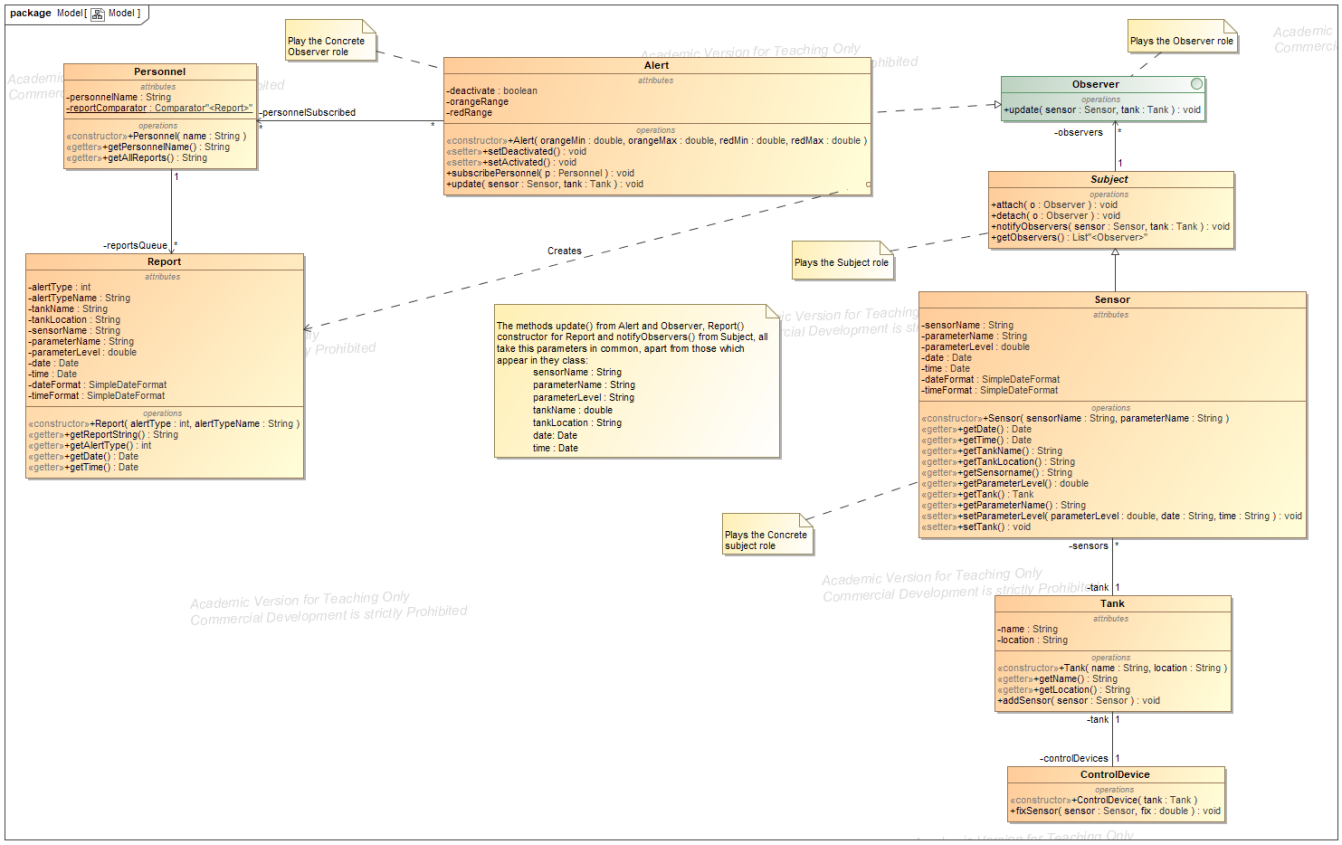
- **Single Responsibility Principle:** in this case we have objects with a single responsibility such as the **Range** class which is only created to check if the parameter which is passed from the **Sensor** to the **Alarm** is inside a stated interval or not with the *contains* method. Another example can also be the report class which is only created to generate objects of the class Report, with the *Report constructor* and which later, will be stored in the *reportsQueue* attribute from **Personnel** class. When following this principle, you are also meeting the "Loose coupling" principle by creating a class such as **Range** which has a single responsibility, the method *contains* and does not depend in any other class.

- **Open Closed Principle:** applied in the **Alert** class which is implementing the **Observer** interface and thus redefining how they want to implement the *update* method. The problem in this case is that when redefining this method, because of the use of the push model when building the observer pattern (explained below) it must pass all the same parameters which are set in the **update** method definition inside the interface. However, the function which this *update* method implements can be redefined independently in each class which implements this Observer interface, so the principle is followed.

- **Liskov substitution Principle**: for example, shown in the **Alert** class which implementing the **Observer** interface and overrides the *update* method, changing its behavior but not contradicting the LSP as the behavior of this method can be expected by the one who calls it in advanced. Also, by the **Sensor** class extending the **Subject** class and thus allowing us to substitute the base class: Subject, by the subclass: Sensor, if necessary as it has the methods necessary just by inheriting them from the base class.

- **Dependency Inversion Principle:** followed by objects from the **Alert** class, which depend on the **Observer** interface making it possible to use any of its methods, even if in this case it is just one method, *update*. Also followed through the exercise when creating attributes such as the *sensors* list in the **Tank** class which wants to use a **List** called *sensors* and then implementing it as an **Arraylist** which is a specific implementation of **List** which can be replaced by other type of list at any time. And the same happens with the *reportsQueue* from the **Personnel** class which is declared as a **Queue** and then implemented as a **PriorityQueue**, making possible to modify the specific implementation of the **Queue** you want. Also, there is a Dependency injection by constructor injections and setter injections in some classes such as in the **Sensor** class in which this class is created passing the attribute *sensorName* as an argument and setting parameters like *parameterLevel* with a setter.

CLASS DIAGRAM:



DYNAMIC DIAGRAM: sequence diagram