SECOND REPORT:

In the second exercise of this assignment we had use as main pattern the observer pattern which is allows one object (the subject, in this case the sensors) to notify multiple other objects (the observers, in this case the ControlDevices) when its state changes.

This is useful when you want to keep multiple objects in synchronous with each other, without having to update each object individually.

We decided to use the pull model, because the subject sends nothing but the most minimal notification, and observers ask for details explicitly. As an advantage, the subject is generally oblivious to the needs of the observers.

As a disadvantage we have that the observers must determine what has changed, but we are making Observers and Subject more independent.

In our code, the observer pattern is used to implement a system for monitoring and controlling the conditions in the aquarium. The Aquarium class has a list of tanks and personal, and methods for changing sensor values and showing the reports when needed, the Aquarium class notifies the ControlDevice class, which make the fired sensor value to be the default value which is considered the first value of the sensor when it was created, they also generate a report and adds it to a queue. The Personal class can then view these reports and see the current state of the aquarium.

The code also follows several principles of software design:

The Single Responsibility Principle (SRP) says that a class should have only one reason to change. In our code, each class has a single, well-defined responsibility. For example, the Aquarium class is responsible for managing tanks and personal, the ControlDevice class is responsible for observing sensors and generating reports, and the Personal class is responsible for subscribing to sensors and viewing reports. This makes the code easier to understand, because each class has an specific purpose and doesn't try to do too much.

The Don't Repeat Yourself (DRY) principle says that you should avoid repeating the same code in multiple places. In the provided code, the Aquarium class has a method called attDevs() that attaches control devices to all sensors in a given tank, which avoids the need to manually attach control devices to each sensor individually. This helps to reduce duplication, because if you need to change how control devices are attached, you only need to modify the attDevs() method, rather than having to update multiple places in the code.

The Keep It Simple, Stupid (KISS) principle says that you should try to keep your code as simple as possible, without making it complex to read. In our code, the classes are relatively small and simple, which makes them easy to understand and modify. For example, the Aquarium class has only six methods, and each method has a clear purpose. This makes the code more maintainable,

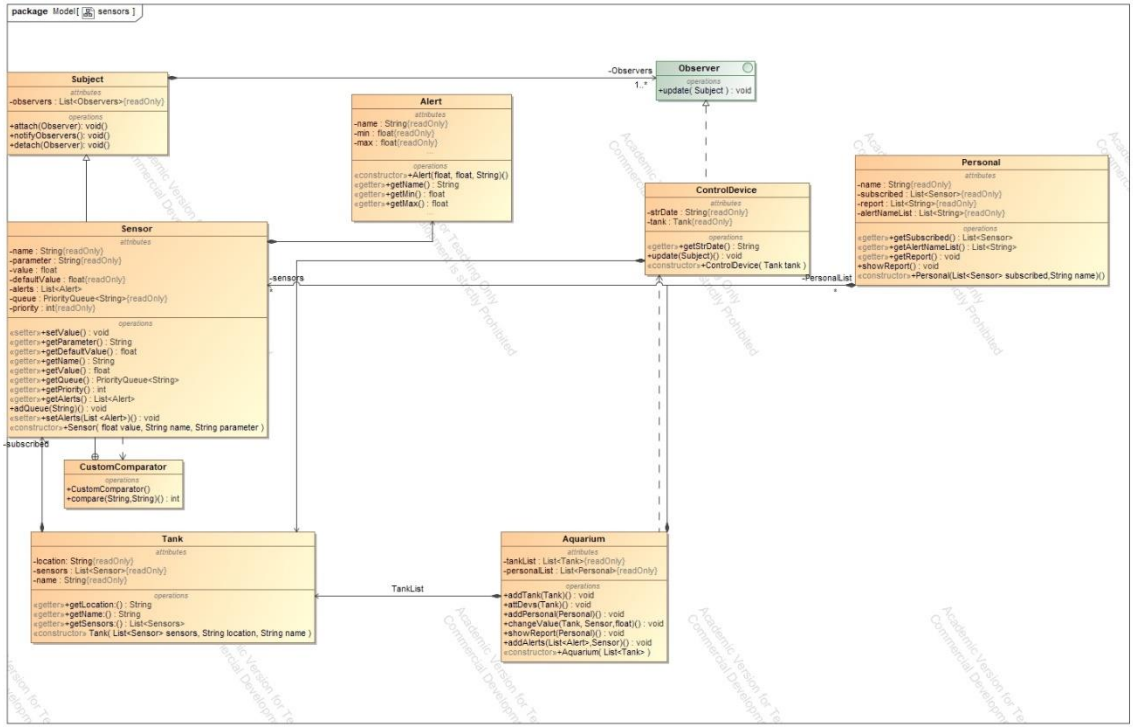
because it's easier to understand and modify, and it's less probable to contain errors.

The You Aren't Gonna Need It (YAGNI) principle states that you should avoid implementing features or functionality that you don't currently need. In the provided code, there are no unnecessary features or unused code, which helps to keep the code focused and maintainable. For example, the Alert class only has the fields and methods that are required to represent an alert, and it doesn't include any unnecessary function. This makes the code easier to understand and maintain, because it only includes the functionality that is actually needed.

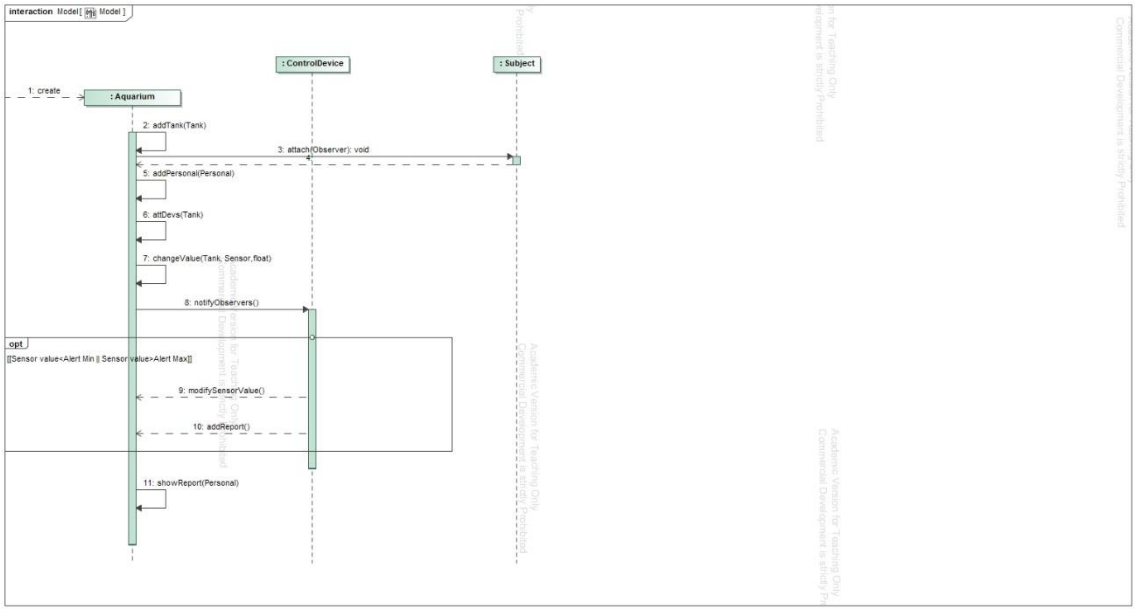
In summary, our code is an example of implementation using the observer pattern, and it follows some principles of good software design. By using the observer pattern, the code allows multiple objects to be notified when the state of the subject changes, and by following good software design principles, the code is easy to understand, maintain and read.

The corresponding diagrams are the following ones:

-Class diagram:



-Sequential diagram:


interaction Model[Model]
: ControlDevice
: Subject
1: create
: Aquarium
2: addTank(Tank)
3: attach(Observer): void
5: addPersonal(Personal)
6: attDevs(Tank)
7: changeValue(Tank, Sensor, float)
8: notifyObservers()
opt
[[Sensor value<Alert Min || Sensor value>Alert Max]]
9: modifySensorValue()
10: addReport()
11: showReport(Personal)