Exercise: Hospital Bed, GoF Strategy

In these exercises, you will start working on a Hospital Bed implementation. You will start out with some code, which in the coming weeks will be replaced by a better design that you will develop.

Today we will extend the solution to use the GoF Strategy pattern to make the software more flexible and even change behavior at runtime.

Your job is to design and implement software for a hospital bed.

The caretakers at the hospital have trouble with patients leaving their beds when they are not supposed to.
The hospital has hired *you*, a brilliant young engineer, to solve their problem, after their old rookie software developer left for fame and money, leaving you with their old legacy code.
The rookie developer left you with a simple solution which meant that:

> Install a presence sensor and a buzzer in the beds.
> If the presence sensor detects, that there is no patient in the bed, alarm the caretakers using the buzzer.

The rookie developer left the hospital bed system when the sensor for detecting the presence or absence of a patient was installed. Luckily the system was prepared for extending the system with a buzzer.

Exercise 1:
Look at the system left by the Rookie developer and draw a UML diagram for the existing system.

Exercise 2:
Create a class that extends `IBedControl` interface, in this class, you can create the logic for the buzzer. Simulate the buzzer by printing to the console when the buzzer is buzzing. Remember to update your UML.

Exercise 3:
Create a Console Application `HospitalBedApplication` and create an instance of `BedControl` and `PresenceSensor` – call the `ReadSample` method to test the application – you can call this method in a loop (infinite or not) to make the system run for an extended time.

Now, you are going to refactor and extend your solution so that it becomes much more flexible. You are also going to implement new ways to alarm caretakers, and finally make it very flexible to select how alarming will take place. This is going to happen by means of the GoF Strategy pattern.

Exercise 4:
Alarming is a candidate for the application of GoF Strategy pattern, so: Design and refactor your application so that alarming uses GoF Strategy. Implement alarming in two ways.

- A buzzer
- A light

How hard would it be to implement a third way of alarming, i.e., a text/email to the caretakers?

Exercise 5:
Redesign your software so that the alarm method i.e., "strategy" can be changed at run-time, so that you can choose the strategy when the program runs. Pressing the 'B' key shall make the program use the buzzer and pressing the 'L' key shall make the program use the light.

Exercise 6:
A test has shown that the sensor sometimes provides false values, i.e., occasionally it reports false, even though a patient is in bed.
As an engineer, you know that you sometimes have to filter your input signals. So, you come up with a very simple filter: The sensor input must be consistent for 3 consecutive samples before it can be trusted. Update your design to

include the filter before the alarm is called. When you do, have the Open Closed Principle in mind. Implement your solution, after you have updated your design.

Proposal: Create a separate class for the filter. The method signature could look like

```
bool Filter(bool sample)
```

where the Filter method is called every time a new sample is read from the sensor. The sample shall be the parameter passed to the method and the method shall return the filtered value. You have to choose, what the initial value should be: absent or present, because you need at least 3 samples to decide the state based on the samples from the presence sensor.

Exercise 7:
The filtering is also a candidate for the application of the GoF Strategy. By applying the GoF Strategy, your software can contain two or more types of filtering:

- Pass-through filter, i.e., no filtering
- Noise filter, i.e., 3 consecutive samples before the input is trusted

Update your design and refactor your software for the application of GoF Strategy, i.e., that it contains both ways of filtering and the chosen filter can be changed while the program runs.
How hard would it be to implement and use a third type of filter?