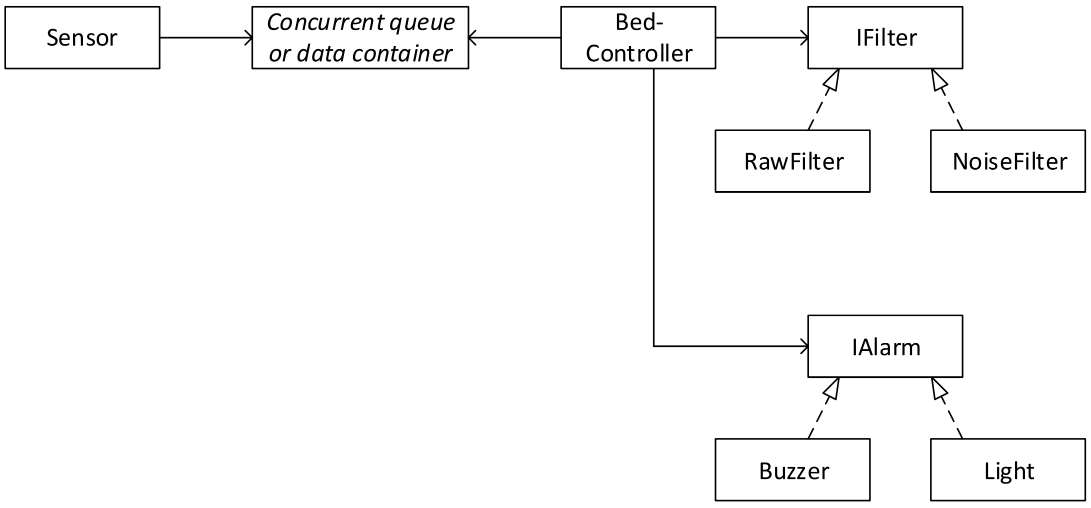
Exercise: Hospital Bed – GoF Observer

In this exercise, you will continue to work with the Hospital bed, the exercise you have already started. We will extend the solution to use the GoF Observer pattern to make the software even more decoupled and extensible.

The last few weeks you worked on the hospital bed system which could detect the presence or absence of a patient in a hospital bed and sound an alarm if the patient left the bed. You have implemented a multi-threaded system using publisher-subscriber, and you have used GoF Strategy to enforce OCP and make the filtering and alarm methods very flexible, even run-time configurable.

Today, you are going to refactor and extend your solution so that it becomes even more flexible and extensible by means of the GoF Observer pattern.

If you have finished the exercise so far, your design probably looks something like the below[1]:



The objective in this exercise is to lower the coupling between the `BedController` and the "alarming", so that the `BedController` does not directly sound or silence the alarm, but instead uses the GoF Observer pattern to notify any listeners that the filter output has changed. One of these listeners is going to be a new class `Alarming` which subscribes to the `BedController` and is thus notified when the filter output changes. Alarming can then sound (or silence) the alarm.

Exercise 1:
Discuss exactly what this exercise requires you to to: How does alarming take place now, and how is it supposed to take place in the future?

Exercise 2:
Suggest a design that uses GoF Observer, so that the BedController can raise a "Patient presence changed" event which `Alarming` can subscribe to and react upon.

Exercise 3:
Implement your design.

Exercise 4:
Extend your design and implementation to introduce a Log class. This class shall log (to screen or file) every time the patient *leaves* his/her bed. Discuss how this new functionality can be introduced into your design. What classes need to change (if any)? Is it easy or hard to introduce this change?

[1] Again: Maybe your classes are called something else. Maybe you have the filter before, not after the queue/data container. Both are just fine – as you know, there is no "right" solution here!

Exercise 5 (advanced, optional)
One disadvantage in the current design is that it does not lend itself to multithreading very well, i.e. if `BedController` and `Alarming` should run in separate threads. Think about it: If “Patient presence changed” occurs, the `BedController` calls `Notify()` on `Alarming`, which then retrieves the state of the patient presence and sounds/silences the alarm. All this, however, occurs in the `BedController`’s thread, since this is the thread that calls `Notify()` in the first place. The intention was that the handling of the event should occur in `Alarming`’s thread.

Your mission, *should you choose to accept it*, is to change the implementation so that the *raising* of the event occurs in the `BedController`’s thread, but the actual *handling* takes place in `Alarming`’s thread. Hint: The registration of the event occurrence and the handling of it is not necessarily the same thing.