**CSE 5322: Software Design Patterns**

**Homework 3**

**Design Problems**

The design problems that are encountered when developing this software are as follows:

1. Since the server object is updated frequently and periodically, there is a need to make this update smoothly and cheaply. Making too many updates for all the patients, all at once could be an expensive task for the hospital.
2. While the server object is being fetched, there is a need to display a placeholder object in place so that the healthcare providers see the last updated value in the worst case. There shouldn’t be a case where they open the software on their device and see no value at all while the software is fetching the latest value from center server.
3. There could be a need to place in a system that can control access to this center server object for privacy reasons. With so many people working at the hospital, there could be different priorities of the access control depending on whether the person is a doctor, or a nurse, or a support staff, or an administrative staff etc.
4. There is a need to broadcast this information to all the people who have registered to receive the warning, where each of them receives uniform information and at the same time.
5. There could be a need to update the list of people who want to receive the notification from a patient, in cases where another doctor or nurse takes over the patient from a different healthcare personnel. The system should then allow easy addition or removal of the healthcare provider.
6. There is also a need to update the system such that when a health care provider receives a warning message, and they act on it, it updates the system such that all the other healthcare providers know that somebody is looking into the patient’s health. So, there is a broadcast of the updated information informing everyone of the status.

**Patterns that solve the problem**

The design patterns that solve the above design problems are:

1. **Proxy pattern**- The first three problems mentioned above can be resolved with the proxy pattern. Here, a proxy class with the same interface as the object in server is defined. The proxy then implements the control and hides the substituted server object.

This helps because the proxy is representing and acting on behalf of the remote object which helps in easy update of the object. It also acts as a placeholder for the server object while the newly updated object is being fetched from the server. This design pattern also helps in implementing control access to the real server object.

1. **Observer pattern**- The last three problem mentioned above can be resolved with the observer pattern. This design pattern helps in easily updating many to one dependencies between the objects, so when the server object changes state (that is when the measurements exceed the limits set by the nurse), all its dependents (that is healthcare providers) are notified and updated automatically.

The benefits of using this design pattern are that it allows easy addition and removal of observer from the list of observers without affecting the observable. This means that if in the future there is a need to add more health care providers, then it can be easily done without changing much in the system. Another benefit of using this design pattern is that it supports multicast and broadcast communication, where the broadcast mechanism is inherited from the observable class. Therefore, this allows easy sharing of the patient’s data with the healthcare providers.

**Design Class Diagram**

Diagram

Description automatically generated

In the Diagram above, there is **Server** that is the center server that receives periodically collected measurements of patient’s health through the device that the patient is wearing. This patient’s data can be accessed by a patients ID.

The **Proxy** class controls access to the server object. It also delegates request to real subject through network communication. Finally, it keeps track of access to the server object so that the resources are released when no other object accesses it.

The **Patient Interface** class defines a common interface for the Server and Proxy classes.

The **PObj** class is a data structure to store information related to the patient which makes it easy to pass it around.

The **Controller** class controls different interactions. It responds to the incoming data from the server. The controller is responsible for issuing warning to the observers as well as requesting the proxy for the patients records.

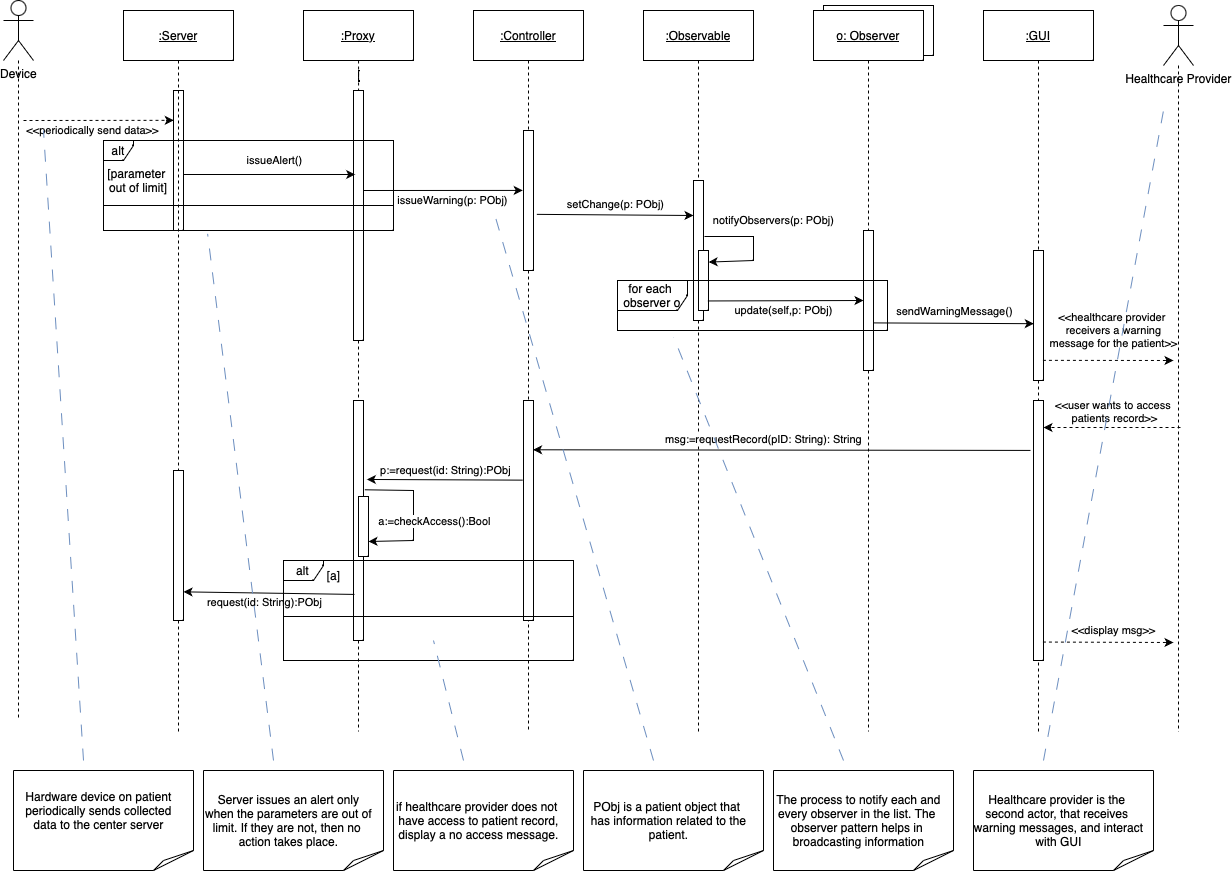
The **Observable** class defines the functions for adding, removing, changing, or notifying the observers. This class also stores a list of observers that have registered to receive this warning. The adding and removing of observers alters this list of observers. The responsibility of this class is to have a knowledge of all the observers in the system.

The **Observer** class is an interface for all concrete observers.

The **Concrete Observer** implements the update method to respond to the change. This is where the all the computers and mobile devices of health care providers receive a notification on their devices. The responsibility of this class is to have knowledge of individual observer.

The **GUI** class is the class that faces the healthcare providers.

**Design Sequence Diagram**



The hardware **device** periodically collects measurements of parameters and sends them to a center server **Server**. The Server calls issueAlert() when any of the parameters falls outside of the limits. The **Proxy** that has a copy of the Server receives the alert and calls issueWarning(p: PObj) with the patient’s information as the parameter. The **Controller** receives this message, and calls observable.setChange(p: PObj). This tells the **observable** to notify the observers, so the Observable calls the notifyObservers(p: PObj). This function then iterates through each **observer** o that has registered to receive the warning, and calls o.update(self, p: PObj). Then each observer, upon receiving the update, sends a warning to the **GUI** by calling sendWarningMessage(). The GUI then shows the healthcare provider the warning message.

The **healthcare provider** can use the **GUI** to access patients record. The GUI calls the **Controller** with the function requestRecord(pID: String): String. This function returns a message that can either contain a patient record or an error message. The controller requests the **Proxy** for patients record by calling the request(id: String): PObj function. The Proxy first checks whether the healthcare provider has access to the record with checkAccess():Bool. If they do, then it fetches the latest information from the **server**, otherwise it returns an error message. The message is then returned to the GUI, after which it is displayed to the healthcare provider.