Observer Pattern

Analysis of Pattern

Observer pattern is very useful when there is state management in application. This pattern basically has subscriber (observer) and Subject (publisher). Subject uses an object to update states coming from observer (members) and update state as initiated by member object or class. Each observer would not have detail of other member's logic but state can be shared in entire application by notifying Subject and back to other observer (member classes). Subject uses observer to notify to child classes about state changes. Observer would update state on object which extend observer.

Benefits

- Makes it easier to manage state in applications
- It's good to implement this in group of objects when there is one to many dependency

Risks of doing this pattern

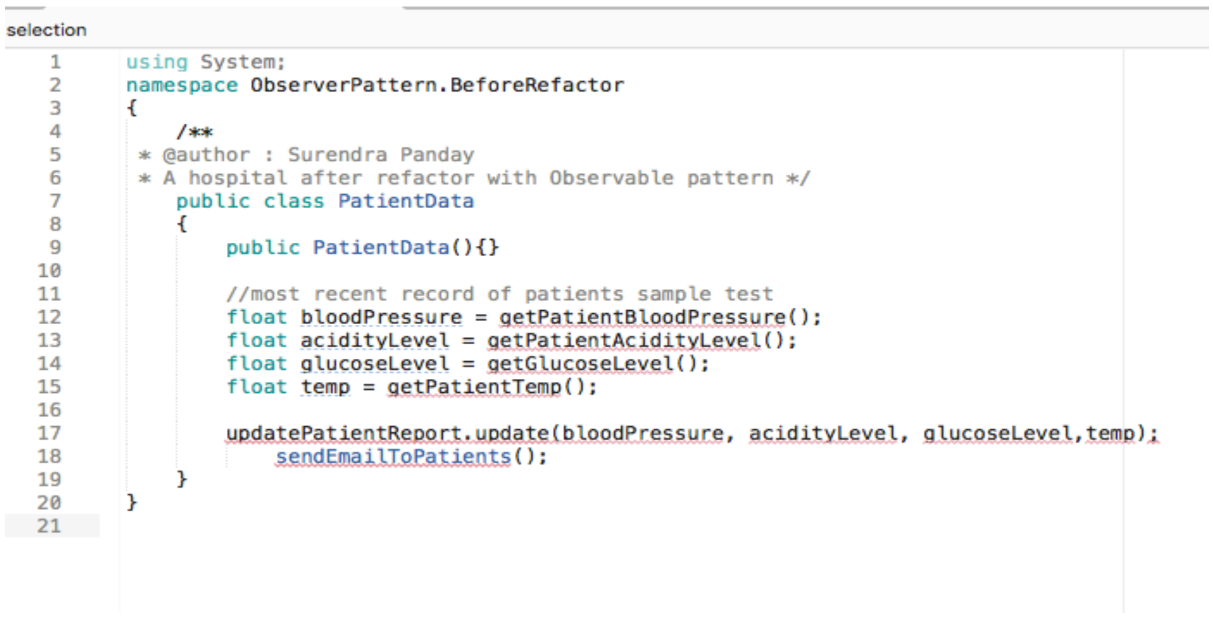
- Observer class is not an interface , it's basically group of objects talking to each other to notify states so code could have duplicate / code smell, it's not as clean as using interface

Scenario

A hospital checks patients' blood pressure, temperature, pulse, height, weight etc. tests and notifies patients about their test results. Patients would not have same data, and they are uniquely identified by their user id, name etc. A hospital would create report for every patient with their statistics from their pressure test, blood test, and so on so forth tests.

With Observable pattern, patients become observer, and hospital becomes subject, and computer system becomes an Object that notifies patients about their test report.

BEFORE REFACTOR

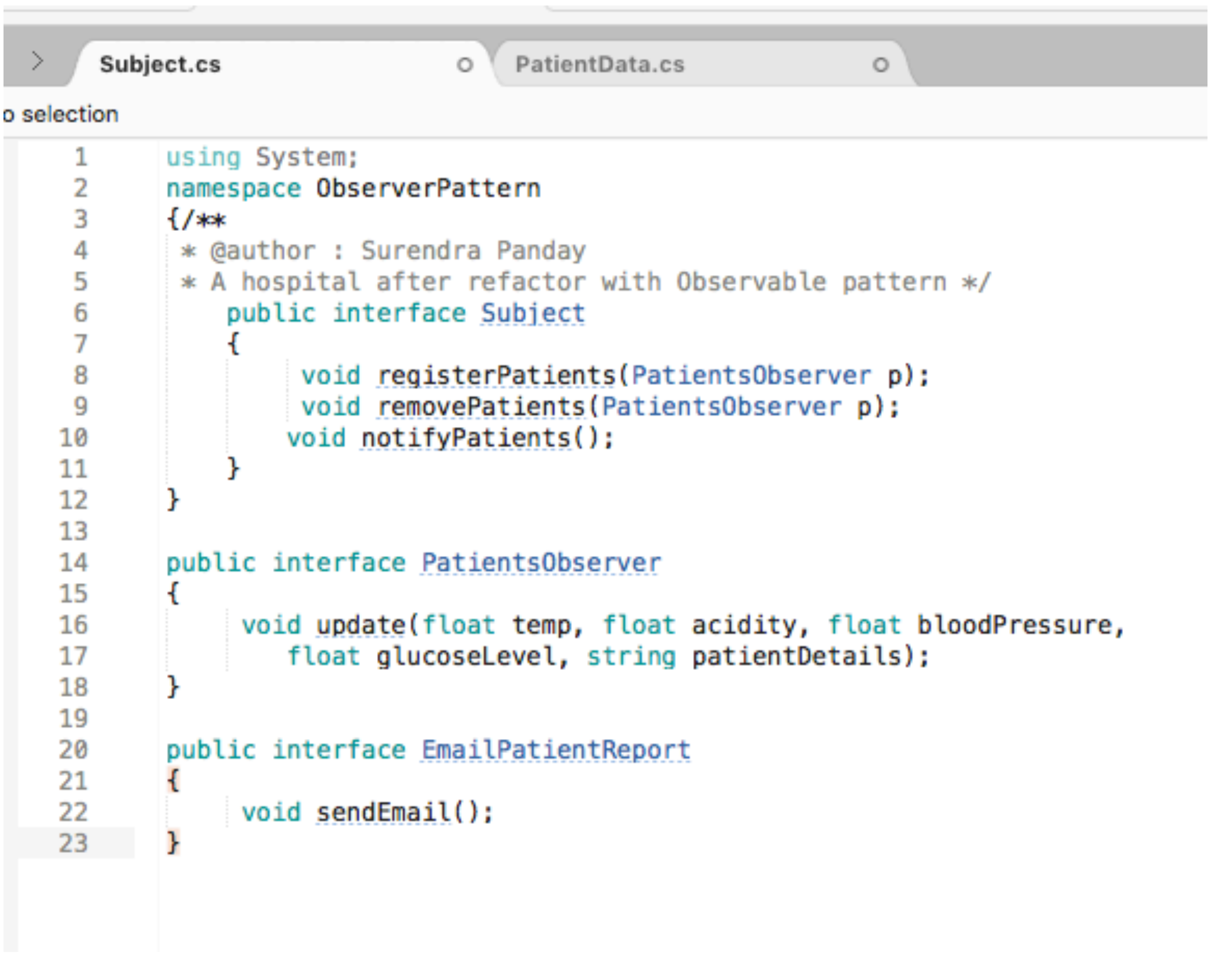


```
using System;
namespace ObserverPattern.BeforeRefactor
{
    /**
 * @author : Surendra Panday
 * A hospital after refactor with Observable pattern */
    public class PatientData
    {
        public PatientData(){}

        //most recent record of patients sample test
        float bloodPressure = getPatientBloodPressure();
        float acidityLevel = getPatientAcidityLevel();
        float glucoseLevel = getGlucoseLevel();
        float temp = getPatientTemp();

        updatePatientReport.update(bloodPressure, acidityLevel, glucoseLevel,temp);
            sendEmailToPatients();
    }
}
```

```
        void sendEmailToPatients();
        void getPatientBloodPressure() { }
        void getPatientAcidityLevel() { }
        void getGlucoseLevel() { }
        void getPatientTemp() { }
```

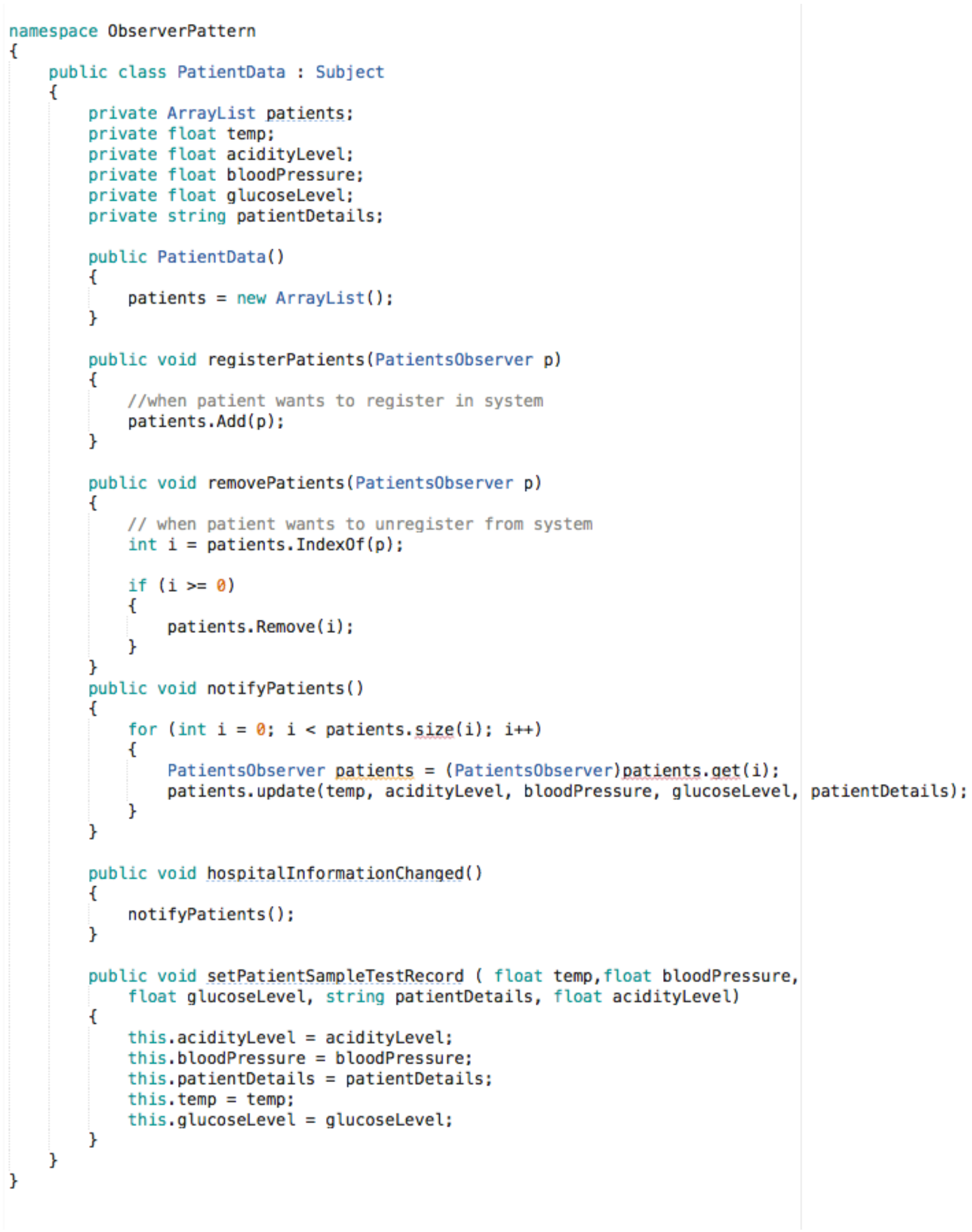
AFTER REFACTOR

A subject that patient data will implement some methods from subject



```
using System;
namespace ObserverPattern
{/**
 * @author : Surendra Panday
 * A hospital after refactor with Observable pattern */
    public interface Subject
    {
         void registerPatients(PatientsObserver p);
         void removePatients(PatientsObserver p);
        void notifyPatients();
    }
}

public interface PatientsObserver
{
     void update(float temp, float acidity, float bloodPressure,
        float glucoseLevel, string patientDetails);
}

public interface EmailPatientReport
{
     void sendEmail();
}
```

A class with observer pattern that implements subject interface



```
namespace ObserverPattern
{
    public class PatientData : Subject
    {
        private ArrayList patients;
        private float temp;
        private float acidityLevel;
        private float bloodPressure;
        private float glucoseLevel;
        private string patientDetails;

        public PatientData()
        {
            patients = new ArrayList();
        }

        public void registerPatients(PatientsObserver p)
        {
            //when patient wants to register in system
            patients.Add(p);
        }

        public void removePatients(PatientsObserver p)
        {
            // when patient wants to unregister from system
            int i = patients.IndexOf(p);

            if (i >= 0)
            {
                patients.Remove(i);
            }
        }
        public void notifyPatients()
        {
            for (int i = 0; i < patients.size(i); i++)
            {
                PatientsObserver patients = (PatientsObserver)patients.get(i);
                patients.update(temp, acidityLevel, bloodPressure, glucoseLevel, patientDetails);
            }
        }

        public void hospitalInformationChanged()
        {
            notifyPatients();
        }

        public void setPatientSampleTestRecord ( float temp,float bloodPressure,
            float glucoseLevel, string patientDetails, float acidityLevel)
        {
            this.acidityLevel = acidityLevel;
            this.bloodPressure = bloodPressure;
            this.patientDetails = patientDetails;
            this.temp = temp;
            this.glucoseLevel = glucoseLevel;
        }
    }
}
```

UML Diagram

check observerPattern.png in ObserverPattern folder

or

https://app.creately.com/diagram/QUaEZAyWDTw/edit