Object Programing – Assignment 2

Objective

The objective of this assignment is to practice implementing the concepts of the inserter, extractor, dynamic memory allocation, inheritance and virtual functions and classes.

Discussion

The use of design patterns is very common in real world programming solutions. A Design pattern is a tried-and-tested architectural framework (in OOA (Object-Oriented Analysis), a collection of interacting objects) that has been proven to robustly solve particular problem types.

A common problem encountered in modern engineering applications is how to ensure that the distributed components of a system always have the most up-to-date information regarding the status of a device under consideration. For instance, consider a security application where you have a remote security system monitoring a house. The system contains motion detectors, trip switches on the windows, etc. You would expect that the security system would automatically inform an arbitrary number of individuals that an event has occurred. The issue here is not whether or not it is possible to maintain a list of individuals, but rather how do you write software that will handle an 'infinite' number of individuals efficiently and robustly without you having to modify the code at some point in the future to facilitate the addition of extra users to the system.

You also may have different categories of homeowners. The security system may be only one part of a bigger home automation system. Some homeowners may not have the same rights to the security system control, but they all should be notified in the event of a break-in…

There are two issues here:

1. How do you maintain a list of unknown/unpredictable numbers of individuals (data structures)? You could use a statically-allocated array (normal allocation). The problem is that the array size must always be worst case, even though you may only require the maximum storage .001% of the time! A standard library exists with optimised algorithms for dealing with situations like this – it is called the Standard Template library (STL). We will use the *vector* class of the STL to assist us in solving this problem – see Section 1.
2. How do you structure your program such that the addition of extra homeowners can be dealt with in a systematic way that does not required subsequent adjustments of the code? Design patterns are tried-and-tested architectural approaches to solving various categories of problems. One such design pattern is the 'Observer' (There are many others!). Section 2 will discuss the Observer design pattern in detail and demonstrate its use.

2. Identifying a Design Pattern to Use

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalised best practices that the programmer can use to solve common problems when designing an application or system[1].

1: Addressing the design problems related with 'Individual Notification'

First of all, take the specification related to notifying an arbitrary number of individuals. You need to design a framework such that when the state (current status of sensors) has changed; all the individuals

[1] source: Wikipedia

(homeowners) are notified regarding the new state (specific location of attempted burglary, window, door, *etc*.) of the house. Now, let us generalize the problem.

Specific Design Problem: *"When the status of the house changes, all the homeowners should be notified immediately."*

Problem Generalized: *"When a subject (in this case, the house sensors) changes, all its dependents (in this case, the homeowners) are notified and updated automatically."*

Once you have such a design problem, you refer to the Gang of Four (GOF) design patterns book[2]- and suddenly you may find out that you can apply the 'Observer' pattern to solve the problem.

Observer Pattern: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

In this case, we used this pattern because we need to notify all the homeowners, when the state of a sensor has changed.

2. Applying the 'Observer' Pattern

In this section, we will have a closer look at the observer pattern, and then we will apply the pattern to solve our first design problem. If you can remember, our first design problem was,

"When the status of any of the sensors of the house changes, all the homeowners and keyholders should be notified immediately."

Understanding the 'Observer' Pattern

The UML class diagram of the observer pattern is shown below.

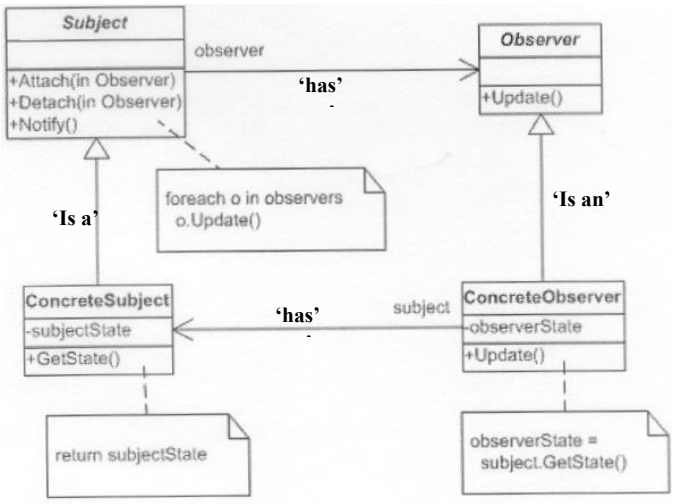

Figure 2: Observer Pattern

The participants of the pattern are detailed below.

[2] '*Design Patterns: Elements of Reusable Object-Oriented Software*' by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Subject

This class provides an interface for attaching and detaching observers. Subject class also holds a private list of observers. Functions in Subject class are

- ***Attach -*** *To add a new observer to the list of observers observing the subject*
- ***Detach -*** *To remove an observer from the list of observers observing the subject*
- ***Notify -*** *To notify each observer by calling the Update function in the observer, when a change occurs.*

ConcreteSubject

This class provides the state of interest to observers. It also sends a notification to all observers, by calling the Notify function in its super class *i.e.* in the Subject class. Functions in the ConcreteSubject class are

- ***GetState -*** *Returns the state of the subject.*

Observer

This class defines an updating interface for all observers, to receive update notification from the subject. The Observer class is used as an abstract class to implement concrete observers

- ***Update -*** *This function is an abstract function, and concrete observers will override this function.*

ConcreteObserver

This class maintains a reference (pointer) to the subject, to receive the state of the subject when a notification is received.

- ***Update -*** *This is the overridden function in the concrete class. When this function is called by the subject, the ConcreteObserver calls the GetState function of the concrete subject to update the local information it possesses regarding the ConcreteSubject's state.*

Adapting the Observer Pattern

Now, let's see how the Observer pattern can be adapted to solve a particular problem. Consider the case of a home security system where the homeowners and key holders need to be informed about changes in the status of the security system.

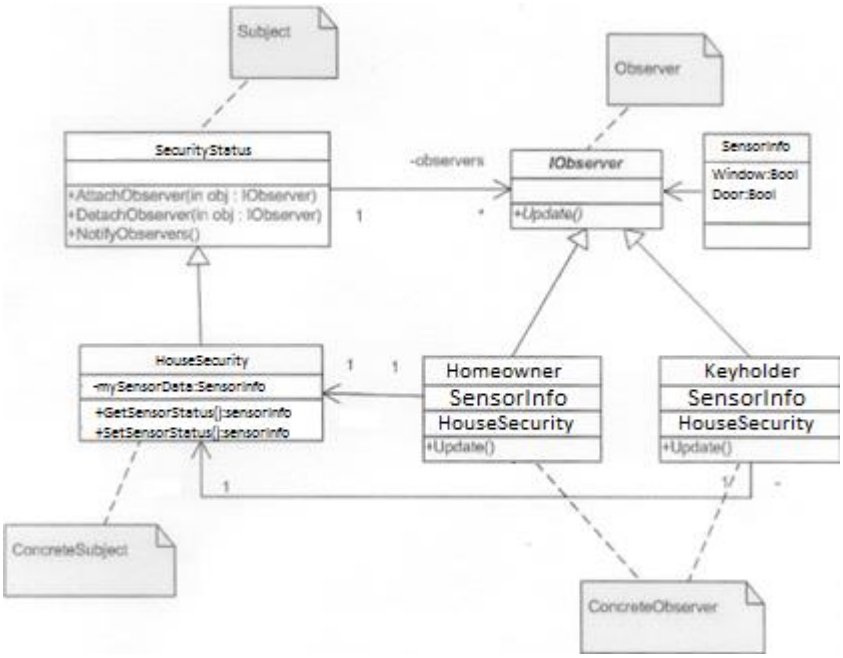

Figure 3: Solving the design problem

When we call the `SetSensorStatus` function of the `HouseSecurity` class to set the new window status it, in turn, calls the `Notify` function defined in the `SecurityStatus` class. The `Notify` function iterates over all the observers in the list, and invokes the `Update` function for each. When the `Update` function is invoked, the observers will obtain the new state of the sensors, by calling the `GetSensorStatus` function in the `HouseSecurity` class.

The participants are detailed below.

IObserver (Observer) *[3 marks]*

```
' Observer: The IObserver Class
'This class is an abstract class. Each homeowner / keyholder must update their
sensor status information from using this method using the SecurityStatus pointer
Abstract Class IObserver

'This method is a must override method                                    [4 marks]
Public pure virtual Sub Update()

End Class ' END CLASS DEFINITION IObserver
```

SecurityStatus (Subject)

The implementation of `SecurityStatus` class is shown below. This class provides interface specifications for creating concrete observers.

```
'Subject : The SecurityStatus Class                                      [3 marks]
Public Class SecurityStatus

      'A private list of observers
      '(Note: you will need to cast the homeowner/ keyholder to an IObserver
      pointer)
      Private observers As STL vector (of IObserver Pointers)            [5 marks]
```

```
        'Routine to attach an observer                                  [5 marks]
        Public Sub AttachObserver(obj As IObserver pointer)
               observers.Add(obj)
        End Sub

        'Routine to remove an observer                                  [5 marks]

        Public Sub DetachObserver(obj As IObserver pointer)
               observers.Remove(obj)
        End Sub

        'Routine to notify all observers                                [8 marks]

        Public Sub NotifyObservers()
           Dim o As IObserver
           For Each o In observers
               o.Update()
           Next
        End Sub
End Class ' END CLASS DEFINITION SecurityStatus
```

HouseSecurity (Concrete Subject)

The implementation of the HouseSecurity class is shown below.

```
' ConcreteSubject : The HouseSecurity Class                            [5 marks]
Public Class HouseSecurity
Inherits SecurityStatus

        'State: The Status of the sensors
        Private mySensorStatus As SensorInfo

        'This function will be called by observers to get current Status of the
        sensors
        Public Function GetSensorStatus () As SensorInfo               [4 marks]
            Return mySensorStatus
        End Function

        'Some external client will call this to set the sensor's position  [4 marks]
        Public Function SetSensorStatus(ByVal defaultStatus As SensorInfo)
               mySensorStatus = defaultStatus

               'Once the sensor status is updated, we have to notify observers
               NotifyObservers()
        End Function

        'Remarks: This can also be implemented as a get/set property
End Class ' END CLASS DEFINITION HouseSecurity
```

SensorInfo Class

We also have a SensorInfo class, to hold the status of the house sensors.

```
' SensorInfo: This is a data structure to hold the status of the house sensors
Public structure SensorInfo                                            [4 marks]
        private window1 As Boolean ' You could use an STL vector here as well…
        private window2 As Boolean
        private door As Boolean

        'This is the constructor
        Public Sub New(Optional ByVal window1  As Boolean = False, _   [5 marks]
            Optional ByVal window2 As Boolean = False, _
            Optional ByVal door As Boolean = False)
            Me.window1 = window1
            Me.window2 = window2
            Me.door = door
        End Sub
```

```
End Class ' END CLASS DEFINITION SensorInfo
```

HomeOwner (ConcreteObserver)

The implementation of the `HomeOwner` class is shown below. `HomeOwner` is inherited from the `IObserver` class.

```
' ConcreteObserver: The HomeOwner Class
inherits from IObserver, and overrides Update method                    [5 marks]
Public Class HomeOwner
Inherits IObserver

       'This variable holds the current state of the sensors            [4 marks]
       Private currentSensorState as SensorInfo

       'These variables hold the personal information of the Homeowner
       Private myPhoneNumber As string
       Private myEmail As string
       Private myName As String

       'This is a pointer to the HouseSecurity in the system
       Private house As HouseSecurity pointer

       'Update() is called from Notify function, in the HouseSecurity class
       Public Overrides Sub Update ()                                   [6 marks]
              currentSensorState = house.GetSensorStatus()
              System.Console.WriteLine("Homeowner {0} says that the status of
              windows 1, 2 and 3is {1},{2},{3} ", _
              myName, currentSensorState.window1, currentSensorState.Window2,
              currentSensorState.door)
       End Sub

       'A constructor which allows creating a pointer to a HouseSecurity object
       Public Sub New(ByVal home As HouseSecurity pointer, ByVal homeOwnerName As
       String, phone as string, email as string)                        [5 marks]
              house = home
              myName = homeOwnerName
              PhoneNumber = phone
              myEmail = email
       End Sub
End Class ' END CLASS DEFINITION HomeOwner
```

Keyholder (ConcreteObserver)

The implementation of the `Keyholder` class is shown below. `Keyholder` is inherited from the `IObserver` class.

```
' ConcreteObserver : The keyholder Class                                [5 marks]
Public Class keyholder
Inherits IObserver

       'This variable holds the current state of the sensors
       Private currentSensorState as SensorInfo

       'These variables hold the personal information of the Homeowner
       Private myPhoneNumber As string
       Private myEmail As string
       Private myName As String

       'This is a pointer to the HouseSecurity object
       Private house As HouseSecurity pointer
       'Update() is called from Notify function, in HouseSecurity class
       Public Overrides Sub Update ()                                   [5 marks]
              currentSensorState = house.GetSensorStatus()
              System.Console.WriteLine("Keyholder {0} says that the status of
              windows 1, 2,and 3 are {1},{2},{3} ", _
```

```
            myName, currentSensorState.window1, currentSensorState.Window2,
            currentSensorState.door)
      End Sub

      'A constructor which allows creating a pointer to a HouseSecurity object
      Public Sub New(ByVal home As HouseSecurity pointer, ByVal KeyholderName As
      String, phone as string, email as string)                          [5 marks]
            house = home
            myName = KeyholderName
            PhoneNumber = phone
            myEmail = email
      End Sub
End Class ' END CLASS DEFINITION KeyHolder
```

Putting It All Together

Now, let's create a `HouseSecurity` object with a few observers. We will attach these observers to the `HouseSecurity` object, so that they are notified automatically when the status of the `HouseSecurity` system changes. The code is pretty self-explanatory.

Consider using smart pointers for the observers…

```
'Let us create a HouseSecurity object and a few observers                [10 marks]

Sub Main()

      'Create our HouseSecurity object (i.e. the concrete subject)
      Dim mySecureHome As New HouseSecurity()

      'Create few HomeOwners (i.e. ConcreteObservers)
      Dim Mum As New Homeowner(mySecureHome,"Mum","123456","Mum@home.ie")
      Dim Dad As New Homeowner(mySecureHome,"Dad","2334567","dad@home.ie")
      Dim Child As New Homeowner(mySecureHome,"Child","0987654","child@home.ie")

      'Create few keyholders (i.e. ConcreteObservers)
      Dim Neighbour1 As New Keyholder(mySecureHome,"Neighbour1","567890","n1@home.ie")
      Dim Neighbour2 As New Keyholder (mySecureHome,"Neighbour2","754321","n2@home.ie")

      'Attach the observers to the house sensor net
      mySecureHome.AttachObserver(Mum)
      mySecureHome.AttachObserver(Dad)
      mySecureHome.AttachObserver(Child)
      mySecureHome.AttachObserver(Neighbour1)
      mySecureHome.AttachObserver(Neighbour2)
      System.Console.WriteLine("After attaching the observers...")

      'Update the status of the house.
      'At this point, all the observers should be notified automatically
      mySecureHome.SetSensorStatus(New sensorInfo())

      'Just write a blank line
      System.Console.WriteLine()

      'Remove some observers
      mySecureHome.DetachObserver(Mum)
      mySecureHome.DetachObserver(Dad)
      System.Console.WriteLine("After detaching Mum and Dad...")

      'Update the status of the sensors again
      mySecureHome.SetSensorStatus(sensorInfo (true, false, false))
      'At this point, all the observers should be notified automatically
      The remaining observers should automatically be notified of the status
      'changes.
      'Press any key to continue…
      System.Console.Read()
End Sub
```

100 marks