Tori Lentz

Dave Retterer

Design Patterns

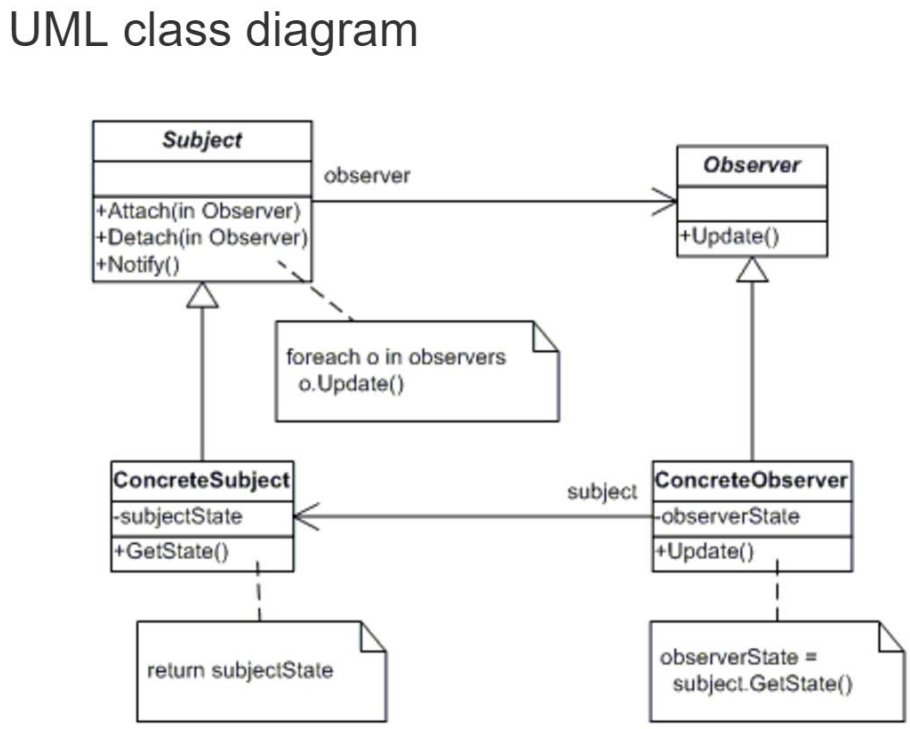
November 21, 2016

Observer Pattern

Introduction

The purpose of this assignment is to design and implement the Observer pattern. This pattern is most often used to notify objects when another object changes state but, it also has the capabilities to add more observers, one for each object you want to watch and see if it changes.

UML Diagram



This pattern is made up of several different parts including: the Subject, Observer, Concrete Subject, and the Concrete Observer. The Subject class knows about all of its observers and any of the observer objects that may observe a subject. This class also provides an interface that allows for the options of attaching and detaching observer objects. Next, is the Concrete Subject which stores the state of interest for each of the Concrete Observers and will send a notification to its observers when its state changes. Thirdly, is the Observer which defines an updating interface for the objects that need to be notified of these different changes in one of the subjects. Finally is the Concrete Observer class which maintains a reference to the Concrete Subject object, will store the state that should stay consistent with the subjects state and, implements the interface that has the updating observer. This part will also keep its state consistent with the subjects.

Code and Description

Subject

```
public abstract class Sender    //Abstract subject
{
    public string _message;
    public string _subject;
    public string _mailTo;
    private List<Observer> _observer = new List<Observer>();

    // Constructor
    public Sender(string message, string subject, string mailTo)
    {
        this._message = message;
        this._subject = subject;
        this._mailTo = mailTo;
    }

    public void Attach(Observer observer)
    {
        _observer.Add(observer);
    }

    public void Detach(Observer observer)
    {
        _observer.Remove(observer);
    }

    public void Notify()
    {
        foreach (Observer observer in _observer)
        {
            observer.Update(this);
        }
        Console.WriteLine("");
    }

    // Gets or sets the email mail to person
    public string mailTo
    {
        get { return _mailTo; }
        set
        {
            if (_mailTo != value)
            {
                _mailTo = value;
                Notify();
            }
        }

    }

    // Gets or sets the email subject
    public string subject
    {
        get { return _subject; }
        set { _subject = value; }
```

This is my sender class, within it I have my three objects; to get my message, subject and mail to variables. Within the methods for these objects I wrote it to get and set the values, as well as, once the mail to value is set, it will notify the observers so the teachers will know that the email has been sent.

```
        }

        // Gets or sets the email message
        public string message
        {
            get { return _message; }
            set {_message = value; }
        }
    }
```

Concrete Subject

```
    public class ConcreteSender : Sender    //Concrete Subject
    {
        // Constructor
        public ConcreteSender(string message, string subject, string mailTo)
          : base(message, subject, mailTo)
        {
        }
    }
```

This is my concrete sender class which sets up the base for my other classes.

Observer

```
    public interface Observer   //Observer interface
    {
        void Update(Sender sender);
    }
```

Here is my observer class and, within it, there is only one method which will update the observers once the objects have been changed.

Concrete Observer

```
    public class ConcreteObserver : Observer //Concrete Observer
    {
        private string _name;
        private Sender _sender;

        // Constructor
        public ConcreteObserver(string name)
        {
            this._name = name;
        }

        public void Update(Sender sender)
        {
            System.Windows.Forms.MessageBox.Show("Observer has been notified:
");
            Console.WriteLine("Notified {0} {1} {2} {3} " , _name, sender._mailTo,
sender._subject, sender._message);
            //this will be a message box and will show name of observer then
sender.Message and will pop up after message has been sent
        }

        // Gets or sets the stock
        public Sender Sender
        {
            get { return _sender; }
            set { _sender = value; }
        }
    }
```

My concrete observer class has several different methods within it, such as, a constructor, the update method, and my sender get and sets are here as well.

Gmail Class

```
    public class Gmail
    {

        public static void SendGmail(string userName, string password, string mailFrom,
string commaDelimCCs, bool isBodyHtml, string mailTo, string subject, string message)
        {

            SmtpClient smtp_server = new SmtpClient("smtp.gmail.com", 587);
            smtp_server.Credentials = new System.Net.NetworkCredential(userName,
password);
            smtp_server.DeliveryMethod = SmtpDeliveryMethod.Network;
            smtp_server.EnableSsl = true;

            MailMessage e_mail = new MailMessage();
            e_mail.From = new MailAddress(mailFrom);
            e_mail.To.Add(mailTo);
            e_mail.Subject = subject;
            e_mail.IsBodyHtml = false;
            e_mail.Body = message;
            smtp_server.Send(e_mail);
        }

    }
```

My Gmail class only deals with the actual sending of the email and getting the text from the text boxes.

Form

```
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {

            ConcreteSender cs = new ConcreteSender("", "", "");
            cs.Attach(new ConcreteObserver("Class Professor"));
            cs.Attach(new ConcreteObserver("Class Assistant Professor"));

            cs.mailTo = tb_MailTo.Text;
            cs.subject = tb_Subject.Text;
            cs.message = tb_Message.Text;

            MailUtilities.Gmail.SendGmail(tb_SenderAccount.Text, tb_AccountPassword.Text,
tb_SenderEmailAddress.Text,  "", true, cs.mailTo, cs.subject, cs.message);
        }

        private void tb_AccountPassword_Enter(object sender, EventArgs e)
        {
            // Set to no text.
            tb_AccountPassword.Text = "";
            // The password character is an asterisk.
            tb_AccountPassword.PasswordChar = '*';
            // The control will allow no more than 14 characters.
            tb_AccountPassword.MaxLength = 14;
```

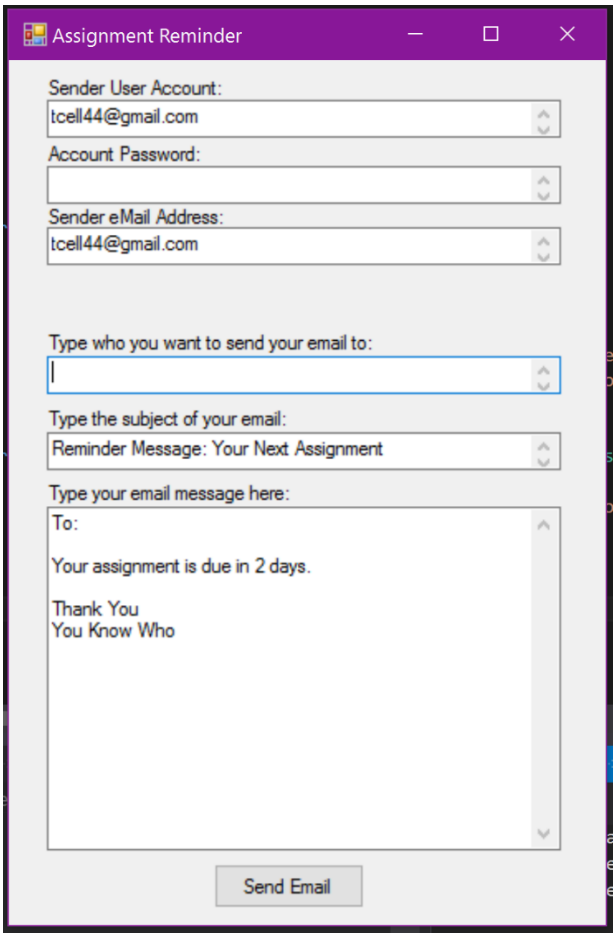
This is my form which consists of a button click action that will send the email and, within this click event, I call all of my necessary elements, as well as, get the text from the textboxes to feed to my other classes.

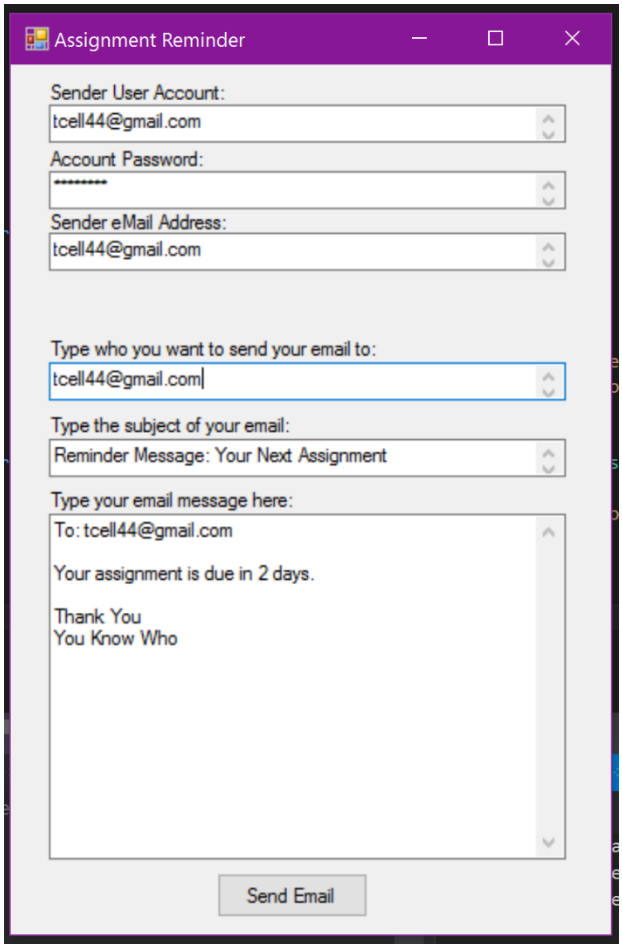
This method will get the entered password and make it so the characters appear as asterisks.

```
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            //setup default assignment reminder message
            tb_SenderAccount.Text = "tcell44@gmail.com";
            tb_SenderEmailAddress.Text = "tcell44@gmail.com";
            tb_Subject.Text = "Reminder Message: Your Next Assignment";
            tb_Message.Text = "To: " + tb_MailTo.Text + "\r\n\r\nYour assignment is due
in 2 days. \r\n\r\nThank You\r\nYou Know Who";
        }

        private void tb_MailTo_TextChanged(object sender, EventArgs e)
        {
            tb_Message.Text = "To: " + tb_MailTo.Text + "\r\n\r\nYour assignment is due
in 2 days. \r\n\r\nThank You\r\nYou Know Who";
        }
    }
```

These next two methods set all the default text for my form.

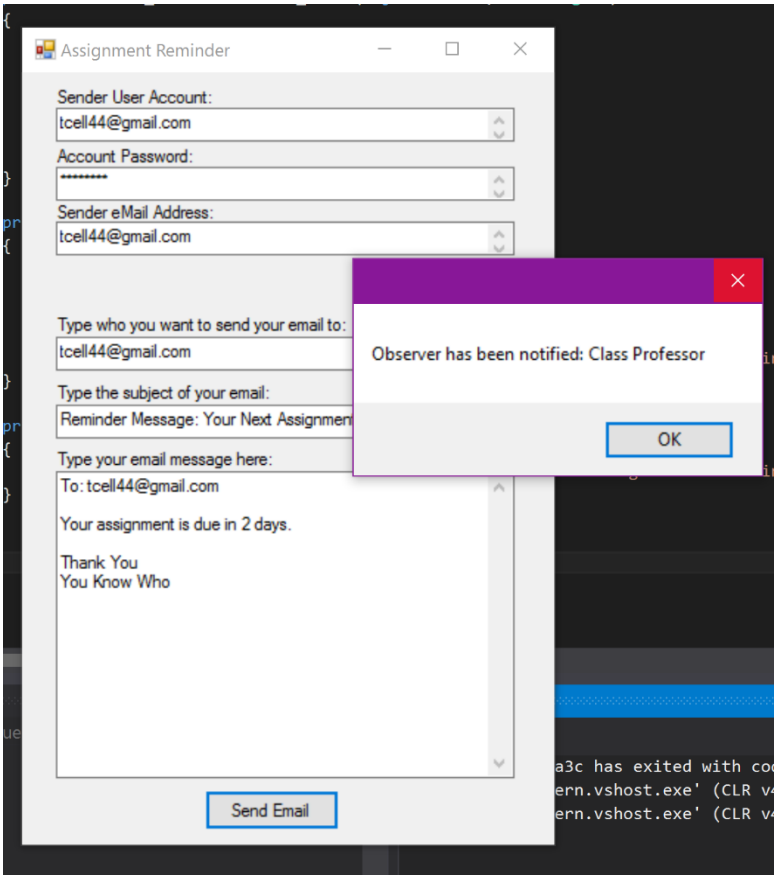
Screen Shots



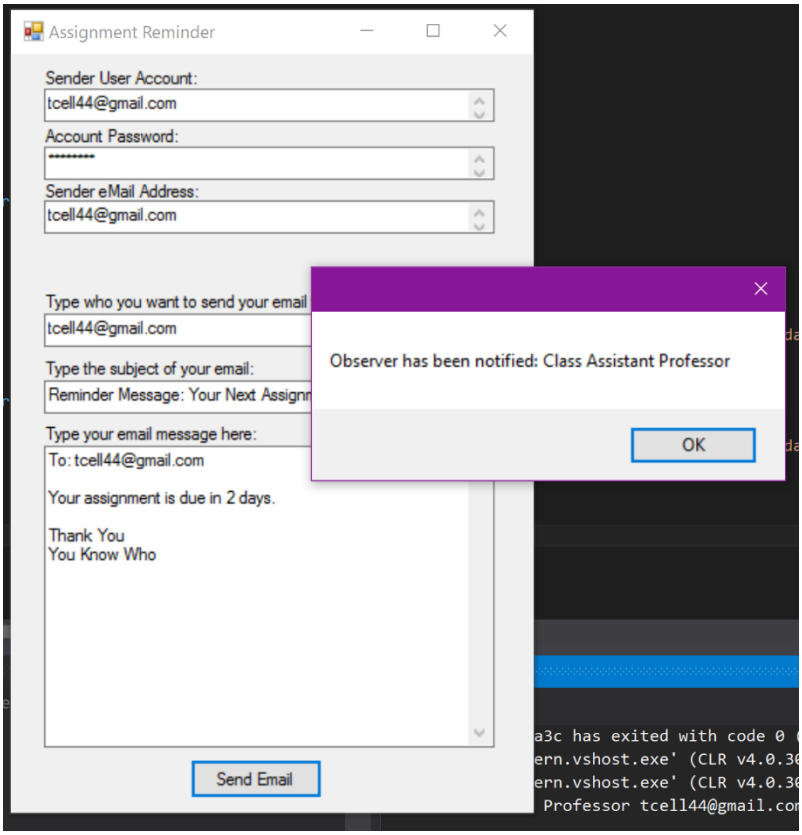


Form on load

Form after button click first observer notified



Form after button click second observer notified



Observations

Overall, this pattern went well. I can see a lot of use for this pattern in a number of different programs. This pattern, I think, would come in handy to check other patters and be used to see how the other program is running. I really liked this program because I was able to do something I had never done before, send something to an outside source.