Accelebrate®

ACCELERATED LEARNING, CELEBRATED RESULTS®

.NET Design Patterns

Lab Manual

https://www.accelebrate.com
(877) 849-1850 ❖ info@accelebrate.com

.NET Design Patterns

C# and Visual Studio

Treeloop
Learning Products

About this Lab Manual

This lab manual provides a series of hands-on exercises for learning about and applying design patterns and best practices in the development of applications built on the Microsoft .NET platform using the C# programming language.

Requirements

- Visual Studio 2012 or newer (any edition except "Express")
- The lab files that accompany this manual

Conventions

Each hand-on exercise in this manual will consist of a series of steps to accomplish a learning objective. Included with the hands-on steps are additional blocks of information identified using one of the following icons:

General information

What needs to be done

What you've accomplished

How things work and why

Where to find more information

Tips and tricks

What not to do

Performance advice

Maintenance advice

Lab 1

Objectives

- Build a Facade over the .NET cryptographic subsystem
- Provide simple methods for encrypting and decrypting strings
  - Use the AES algorithm
  - Return the encrypted value as a Base64 encoded string

Procedures

1. Open the **Cryptotron.sln** solution from the **Lab01\Begin** folder. This solution contains two projects.

   ClientApp is a Windows Forms application that implements a simple interface for providing a string to encrypt or decrypt. Event handlers exist for the two buttons but there is no code yet.

   CryptoLib is a class library that will contain the code for the Facade. This project does not contain any code yet.

2. Add a new **interface** to the **CryptoLib** project named **CryptoFacade** that will define what services the facade will provide.

```
public interface ICryptoFacade
{
  string Encrypt(string plainText);
  string Decrypt(string cipherText);
}
```

3. Create a class that implements the interface and provide an implementation for the two methods. You can copy this code from **Lab01\Code\CryptoFacade.txt**. The code is also provided on the next page.

4. Add code to **ClientApp** so that it uses the functionality provided by the Facade.

```
public class CryptoFacade : ICryptoFacade
{
  private byte[] initVector;
  private byte[] key;
  private AesManaged provider;

  public CryptoFacade()
  {
    provider = new AesManaged();
    initVector = new byte[provider.BlockSize / 8];
    key = new byte[provider.KeySize / 8];
    using (var rngProvider = new RNGCryptoServiceProvider())
    {
      rngProvider.GetBytes(initVector);
      rngProvider.GetBytes(key);
    }
  }

  public string Encrypt(string plainText)
  {
    ICryptoTransform transform = provider.CreateEncryptor(key, initVector);
    byte[] clearBytes = Encoding.UTF8.GetBytes(plainText);
    byte[] cipherBytes;
    using (var buf = new MemoryStream())
    {
      using (var stream = new CryptoStream(buf, transform, CryptoStreamMode.Write))
      {
        stream.Write(clearBytes, 0, clearBytes.Length);
        stream.FlushFinalBlock();
        cipherBytes = buf.ToArray();
      }
    }
    return Convert.ToBase64String(cipherBytes);
  }

  public string Decrypt(string cipherText)
  {
    ICryptoTransform transform = provider.CreateDecryptor(key, initVector);
    byte[] cipherBytes = Convert.FromBase64String(cipherText);
    byte[] clearBytes;
    using (var buf = new MemoryStream())
    {
      using (var stream = new CryptoStream(buf, transform, CryptoStreamMode.Write))
      {
        stream.Write(cipherBytes, 0, cipherBytes.Length);
        stream.FlushFinalBlock();
        clearBytes = buf.ToArray();
      }
    }
    return Encoding.UTF8.GetString(clearBytes);
  }
}
```

5. Test the application my ensure that the encryption operation is reversible.

If your test fails, it might be because you are creating two different instances of the facade. This would result in different keys being used. The facade instance should be defined as a class-level variable in the form.

✓ You have successfully completed Lab 1.

Lab 2

Objectives

- Write a unit test for the Facade created in the previous lab

Procedures

1. If not already open, re-open your solution from Lab 1.

   If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder for Lab 2.

2. Add a new **Unit Test Project** to the solution named **CryptoLibTests**.

3. Add a **reference** to **CryptoLibTests** for **CryptoLib**.

4. Rename **UnitTest1.cs** to **CryptoFacadeTests.cs**. Click **Yes** when asked to rename all references.

5. Write a test that confirms that the encryption is **reversible**. You will need a **using** directive for the **CryptoLib** namespace.

```
[TestClass]
public class CryptoFacadeTests
{
  [TestMethod]
  public void ReversibleEncryption()
  {
    // Arrange
    var cf = new CryptoFacade();
    string str = "This is a test";
    string result;

    // Act
    result = cf.Encrypt(str);
    result = cf.Decrypt(result);

    // Assert
    Assert.AreEqual(str, result);
  }
}
```

6. Ensure the **Test Explorer** window is visible. You can select **[Test > Windows > Test Explorer]** from the Visual Studio menu.

7. In the Test Explorer window, check **Run All**. The test should display as a **pass**.

   It may take a long time for the test to run the first time. This is because the Test Execution Engine takes a while to start. Subsequent runs are much faster since the test engine is already running.

   You have successfully completed Lab 2.

Lab 3

Objectives

- Refactor an existing application to employ the Strategy pattern
- Use a TDD-style approach

Procedures

1. Open the **PropGen.sln** solution from the **Lab03\Begin** folder. This solution contains two projects. One is a class library and the other is a unit test project.

2. **Examine** the existing code.

3. Run the existing **unit tests** and confirm that they all pass.

   The objective for this lab is to modify the project to support different proposal price calculation algorithms. So, the first step will be to define what a proposal price calculation strategy should look like.

4. Add a new **interface** to the class library named **IProposalPriceCalculator** that implements a single method.

```
public interface IProposalPriceCalculator
{
  double CalculateTotalPrice(Proposal proposal);
}
```

5. Move the existing proposal price calculation algorithm into a new class named **StandardProposalPriceCalculator** that implements the interface.

```
public class StandardProposalPriceCalculator : IProposalPriceCalculator
{
  public double CalculateTotalPrice(Proposal proposal)
  {
    double totalPrice = proposal.Course.BasePrice;
    int additionalAttendees = Math.Max(0, proposal.Attendees - 3);
    totalPrice += (additionalAttendees * proposal.Course.PersonPrice);
    return totalPrice;
  }
}
```

6. Add a new strategy named **GsaProposalPriceCalculator** but only provide enough code to get the class to compile successfully.

```
public class GsaProposalPriceCalculator : IProposalPriceCalculator
{
  public double CalculateTotalPrice(Proposal proposal)
  {
    return 0.00;
  }
}
```

The reason for not implementing the logic for this strategy right away is because we are going to write a unit test for this strategy before working on the actual implementation.

7. **Modify** the `CalculateTotalPrice` method of the `Proposal` class to accept and use a strategy.

```
public double CalculateTotalPrice(IProposalPriceCalculator ppc = null)
{
  if (ppc == null) ppc = new StandardProposalPriceCalculator();
  return ppc.CalculateTotalPrice(this);
}
```

Notice that we're using an **optional parameter** to allow for the use of a default strategy if one is not provided. This is not a requirement of the Strategy pattern but is convenient here since it will mean our existing unit tests will still pass.

8. Add a new test method that uses the `GsaProposalPriceCalculator` strategy to calculate the price for a large class.

```
[TestMethod]
public void LargeGsaClass()
{
  // Arrange
  Proposal proposal = new Proposal()
  {
    Course = testCourse,
    Attendees = 15
  };
  IProposalPriceCalculator ppc = new GsaProposalPriceCalculator();

  // Act
  double totalPrice = proposal.CalculateTotalPrice(ppc);

  // Assert
  Assert.IsTrue(totalPrice == 1530.00);
}
```

9. **Run** all the unit tests and confirm that the original tests still pass but the new test **fails** since we haven't implemented that strategy yet.

10. Implement the `CalculateTotalPrice` method of the `GsaProposalPriceCalculator` strategy.

```
public class GsaProposalPriceCalculator : IProposalPriceCalculator
{
  public double CalculateTotalPrice(Proposal proposal)
  {
    double totalPrice = proposal.Course.BasePrice;
    totalPrice += (7 * proposal.Course.PersonPrice);
    totalPrice -= (totalPrice * 0.10);
    return totalPrice;
    }
  }
}
```

11. **Run** the units tests again and confirm that the new test now passes.

✓ You have successfully completed Lab 3.

Lab 4

Objectives

- Add new features to the project from the previous lab using the Decorator pattern

Procedures

1. If not already open, re-open your solution from Lab 3.

   If you do not want to use your solution from the previous lab, you can open the solution in the *Begin* folder for Lab 4.

   In the previous lab, we implement two different price calculation algorithms using the Strategy pattern. In this lab, we will add support for adding expenses to specific proposals using the Decorator pattern.

   The first step will be to define the **abstract** base class for our expense decorators.

2. Add a new **abstract** class to the **PropGenLib** project named **ExpenseDecorator**.

```
public abstract class ExpenseDecorator : IProposalPriceCalculator
{
}
```

3. Add code to `ExpenseDecorator` to support an internal **component** (decorator or price calculator).

```
public abstract class ExpenseDecorator : IProposalPriceCalculator
{
  private IProposalPriceCalculator component;

  public ExpenseDecorator(IProposalPriceCalculator ppc)
  {
    component = ppc;
  }
}
```

   Because `ExpenseDecorator` implements `IProposalPriceCalculator`, it must implement `CalculateTotalPrice`. This method should call the method of it's internal component and then call a method of the decorator to execute the expense calculation logic.

4. Add the two additional methods to `ExpenseDecorator`.

```
public double CalculateTotalPrice(Proposal proposal)
{
  double totalPrice = component.CalculateTotalPrice(proposal);
  totalPrice = AddExpense(proposal, totalPrice);
  return totalPrice;
}

protected abstract double AddExpense(Proposal proposal, double totalPrice);
```

5. Add a new **decorator subclass** named **AirfareExpenseDecorator** that adds **500** to the price of the proposal.

```
public class AirfareExpenseDecorator : ExpenseDecorator
{
  public AirfareExpenseDecorator(IProposalPriceCalculator ppc)
    : base(ppc) { }

  protected override double AddExpense(Proposal proposal, double totalPrice)
  {
    return totalPrice + 500.00;
  }
}
```

6. Add one more **decorator subclass** named **SalesTaxExpenseDecorator** that adds **5%** to the total proposal price.

```
public class SalesTaxExpenseDecorator : ExpenseDecorator
{
  public SalesTaxExpenseDecorator(IProposalPriceCalculator ppc)
    : base(ppc) { }

  protected override double AddExpense(Proposal proposal, double totalPrice)
  {
    return totalPrice + (totalPrice * 0.05);
  }
}
```

7. Add a new test method that checks the price calculated for a **large GSA class** with **airfare** and **sales tax** added.

```
[TestMethod]
public void LargeGsaClassWithAirfareAndSalesTax()
{
  // Arrange
  Proposal proposal = new Proposal()
  {
    Course = testCourse,
    Attendees = 15
  };
  IProposalPriceCalculator ppc = new GsaProposalPriceCalculator();
  ppc = new AirfareExpenseDecorator(ppc);
  ppc = new SalesTaxExpenseDecorator(ppc);

  // Act
  double totalPrice = proposal.CalcuateTotalPrice(ppc);

  // Assert
  Assert.IsTrue(totalPrice == 2131.50);
}
```

8. **Run** the unit tests and confirm that they all pass.

There is nothing wrong with the default site and application pools. It would be alright to use them. For this course, however, we would like to create everything from scratch.

You have successfully completed Lab 4.

Lab 5

Objectives

- Examine an application that uses a manual implementation of the Observer pattern
- Refactor the application to use the .NET event system

Procedures

1. Open the **Eventful.sln** solution from the **Lab05\Begin** folder. This solution contains two projects. One is a class library and the other is a unit test project.

2. **Examine** the existing code.

   These is one unit test method but that method is marked as inconclusive.

   This project currently uses a traditional implementation of the Observer pattern so that the billing service can be notified of when an album is played so that the user can be charged.

3. **Examine** the code in both projects.

   The objective of this lab is to refactor the **EventfulLib** project so that is accomplishes the same goal but using .NET events.

4. Modify the `Album` class no that it no longer inherits from `Subject` and define an **event** in the `Album` class named **PlayEvent**.

```
public event EventHandler PlayEvent;
```

5. Add a **private** method named **Notify** that fires `PlayEvent`.

```
private void Notify()
{
  if (PlayEvent != null)
  {
    PlayEvent(this, EventArgs.Empty);
  }
}
```

6. Modify `BillingService` so that it no longer inherits from `Observer`.

We are no longer using the Subject and Observer classes. You could delete these classes now if desired.

We can now use unit testing to test the `Album` class independent of the `BillingService`. This can include checking for the firing of the **event**.

7. Add a new **Unit Test** to the **EventfulTests** project named **AlbumTests**.

8. Add a private field for an **album** and a **boolean** field that will be used to record if the **PlayEvent** was fired.

```
[TestClass]
public class AlbumTests
{
  private bool eventFired;
  private Album album;
}
```

9. Add a **TestInitialize** method to initialize the private fields.

```
[TestInitialize]
public void Init()
{
  album = new Album("Up");
  eventFired = false;
}
```

10. Add a method that can be used as an **event handler** for `PlayEvent`. For testing purposes, we will have this method simply set a variable so we can easily check if the event handler was in fact called.

```
private void OnPlay(object subject, EventArgs eventArgs)
{
  eventFired = true;
}
```

11. Add a **test method** that attaches an event handler, invokes the `Play` method of an album, and confirms the event handler was called.

```
[TestMethod]
public void Attach()
{
  album.PlayEvent += OnPlay;
  album.Play();

  Assert.IsTrue(eventFired);
}
```

12. Add a second **test method** that invokes the `Play` method without attaching the event handler. In this case, the event handler should not be called.

```
[TestMethod]
public void DoNotAttach()
{
  album.Play();
  Assert.IsFalse(eventFired);
}
```

13. **Run** all of the unit tests. The original test method is still inconclusive but both of the new test methods should pass.

✓ You have successfully completed Lab 5.

Lab 6

Objectives

- Extend an existing application using the Command pattern

Procedures

1. Open the **Automobile.sln** solution from the **Lab06\Begin** folder. This solution contains two projects. One is a class library and the other is a console application.

   The **AutomobileLab** project contains two classes that represent automobile components (`Radio` and `ElectricWindow`). Each has an interface that can be used to control the component.

   The objective of the lab is to extend the application by allowing both devices to be controlled by a new component (`SpeechRecognizer`).

2. **Examine** the existing code.

3. Add a new **interface** named **IVoiceCommand** with an Execute method.

```
public interface IVoiceCommand
{
  void Execute();
}
```

   This interface defines an abstraction for the commands that will be issued by the `SpeechRecognizer` component that we will add next.

4. Add a new class named **SpeechRecognizer**.

```
public class SpeechRecognizer
{
}
```

   The SpeechRecognizer will initially support two different command types (up and down).

5. Add private fields and a constructor to `SpeechRecognizer` to support the two different commands it can execute.

```
public class SpeechRecognizer
{
  private IVoiceCommand upCommand, downCommand;

  public void SetCommands(IVoiceCommand upCommand, IVoiceCommand downCommand)
  {
    this.upCommand = upCommand;
    this.downCommand = downCommand;
  }
}
```

⚙ Instances of `IVoiceCommand` are provided via the constructor so that the `SpeechRecognizer` can easily control different devices via the commands.

⚙ These two commands will be executed when the `SpeechRecognizer` identifies the correct spoken words.

6. Add two methods to the `SpeechRecognizer` class to simulate this.

```
public void HearUpSpoken()
{
  upCommand.Execute();
}

public void HearDownSpoken()
{
  downCommand.Execute();
}
```

7. Create a new class named **VolumeUpCommand** that implements `IVoiceCommand` and calls the appropriate method of a radio instance.

```
public class VolumeUpCommand : IVoiceCommand
{
  private Radio radio;

  public VolumeUpCommand(Radio radio)
  {
    this.radio = radio;
  }

  public void Execute()
  {
    radio.VolumeUp();
  }
}
```

8. Create similar classes for **VolumeDownCommand**, **WindowUpCommand**, and **WindowDownCommand**.

9. Is is now possible to implement **AutomobileClient** so that an instance of `SpeechRecognizer` is used to control a `Radio` instance as well as an `ElectricWindow` instance.

```
static void Main(string[] args)
{
  Radio radio = new Radio();
  radio.SwitchOn();
  IVoiceCommand volumeUpCommand = new VolumeUpCommand(radio);
  IVoiceCommand volumeDownCommand = new VolumeDownCommand(radio);

  ElectricWindow window = new ElectricWindow();
  IVoiceCommand windowUpCommand = new WindowUpCommand(window);
  IVoiceCommand windowDownCommand = new WindowDownCommand(window);

  SpeechRecognizer speechRecognizer = new SpeechRecognizer();
  speechRecognizer.SetCommands(volumeUpCommand, volumeDownCommand);
  Console.WriteLine("Speech recognition controlling the radio");
  speechRecognizer.HearUpSpoken();
  speechRecognizer.HearUpSpoken();
  speechRecognizer.HearUpSpoken();
  speechRecognizer.HearDownSpoken();

  speechRecognizer.SetCommands(windowUpCommand, windowDownCommand);
  Console.WriteLine("Speech recognition controlling the window");
  speechRecognizer.HearDownSpoken();
  speechRecognizer.HearUpSpoken();

  Console.ReadKey();
}
```

10. **Run** the application and confirm the behavior.

✓ You have successfully completed Lab 6.