

Design Patterns

*Adapter Pattern**

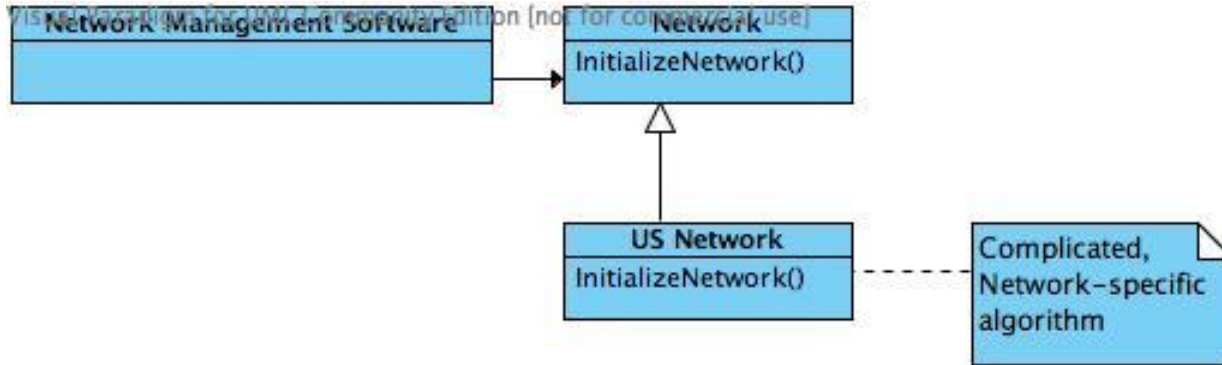
ebru@hacettepe.edu.tr
ebruakcapinarsezer@gmail.com
<http://yunus.hacettepe.edu.tr/~ebru/>
[@ebru176](#)

Kasım 2017



*modified from <http://www.classes.cs.uchicago.edu>

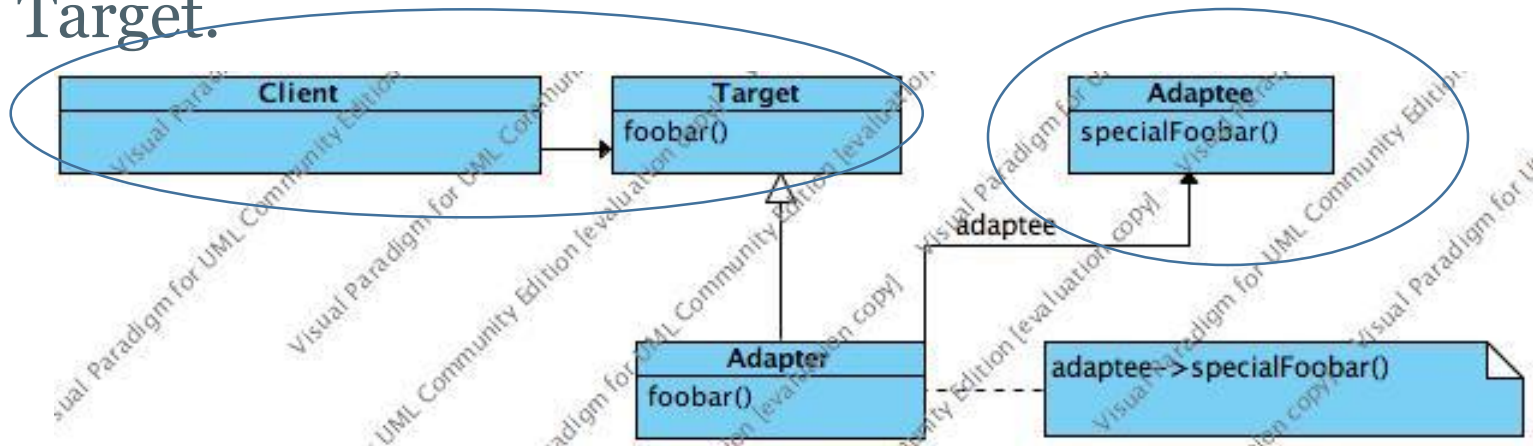
- You program for the control center of a US phone company. Your network management software is Object Oriented with a simple hierarchy...



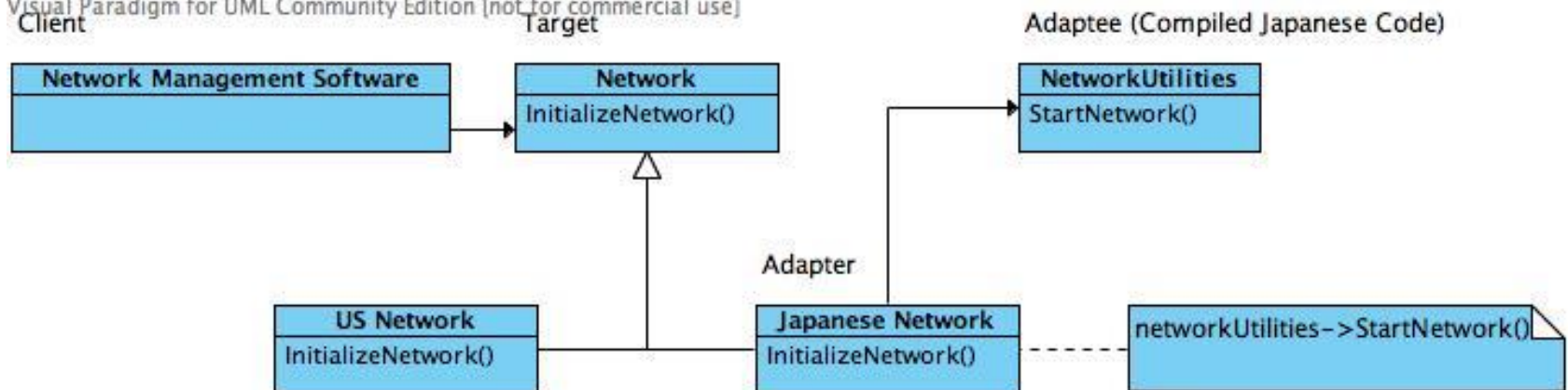
- Your company acquires a Japanese phone company. The workings of a Japanese network are complex and foreign to you, but you need to incorporate this new network into your software.
- The Japanese company used similar software with a **NetworkUtilities** class containing a **StartNetwork** operation rather than **InitializeNetwork**. You are only given a compiled version of the class.
- Do you begin studying Japanese communication protocols and waste millions of company dollars implementing your own **InitializeNetwork** algorithm?

- *The adapter pattern is a design pattern that is used to allow two incompatible types to communicate. Where one class relies upon a specific interface that is not implemented by another class, the adapter acts as a translator between the two types.*
- 3 essentially classes involved:
 - **Target** - class that needs to steal operations from some other class (Adaptee)
 - **Adapter** - class that wraps the operations of the foreign class (Adaptee) in Target-familiar interfaces
 - **Adaptee** - class with operations desired for the Target class
- In Class Adapter, this was accomplished by having the Adapter inherit from both the Target and the Adapter.

- In Object Adapter, the Adapter subclasses the Target. Instead of subclassing the Adaptee, the Adapter internally holds an instance of the Adaptee and uses it to call Adaptee operations from within operations familiar to the Target.



Visual Paradigm for UML Community Edition (not for commercial use)



Network.java - the target

```
public abstract class Network {  
    abstract void InitializeNetwork();  
}
```

JapaneseNetwork.java - the adapter

```
public class JapaneseNetwork extends Network {  
    //instance of the adaptee  
    NetworkUtilities myAdapteeObject;  
  
    public void JapaneseNetwork(NetworkUtilities nu) {  
        myAdapteeObject = nu;  
    }  
  
    public void InitializeNetwork() {  
        myAdapteeObject.StartNetwork();  
    }  
}
```

NetworkUtilities.java - the adaptee

```
public class NetworkUtilities {  
  
    public void StartNetwork() {  
        //complex Japanese network code...  
    }  
}
```

Tester.java - the client

```
public class Tester {  
    public void Tester() {  
        NetworkUtilities nu = new NetworkUtilities();  
        JapaneseNetwork jNetwork = new JapaneseNetwork(nu);  
        USNetwork uNetwork = new USNetwork();  
  
        //both networks use familiar interface but jNetwork  
        uses        //adaptee  
        uNetwork.InitializeNetwork();  
        jNetwork.InitializeNetwork();  
    }  
}
```

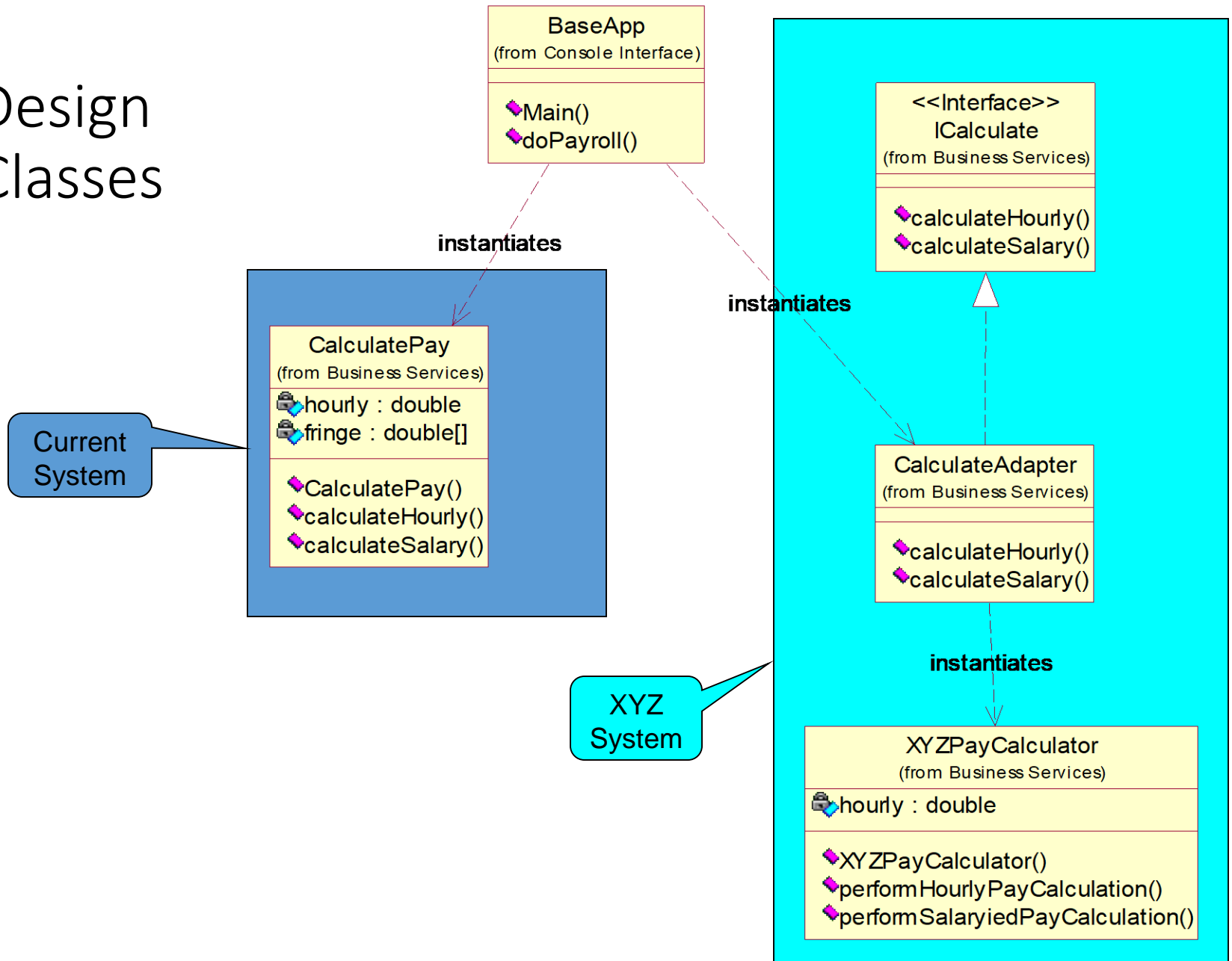
Problem

- Your payroll has been calculated by an internal class. However you have decided to use a third party class that will give you more flexibility.
- The methods used for calculating payroll the third party XYZ class have different names and parameters.
- How is the best way to implement the XYZ class with as little changes as possible.

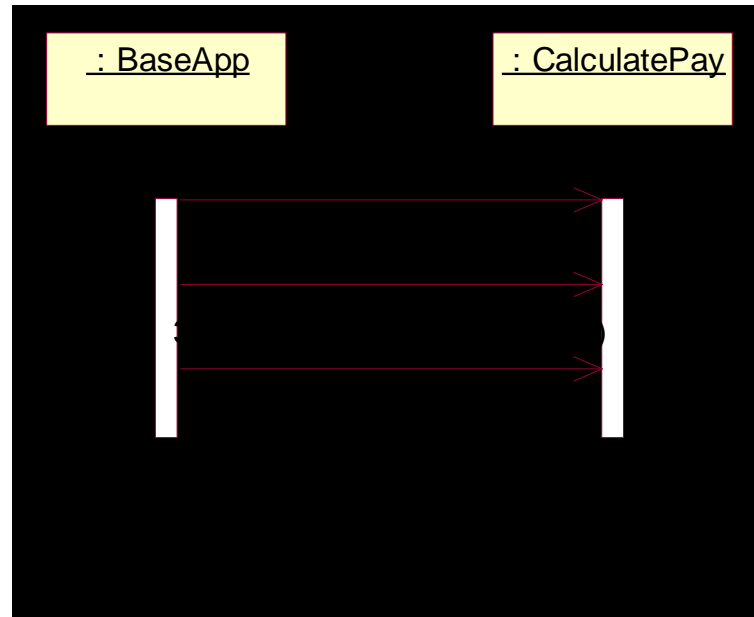
Solution is to Use an Adapter

- Create an interface that defines all the existing method calls to the present calculation class.
- Build an adapter that implements the new interface so that no method calls will be broken
- Implement methods in the adapter that accept the old calls and make the new calls to XYZ.

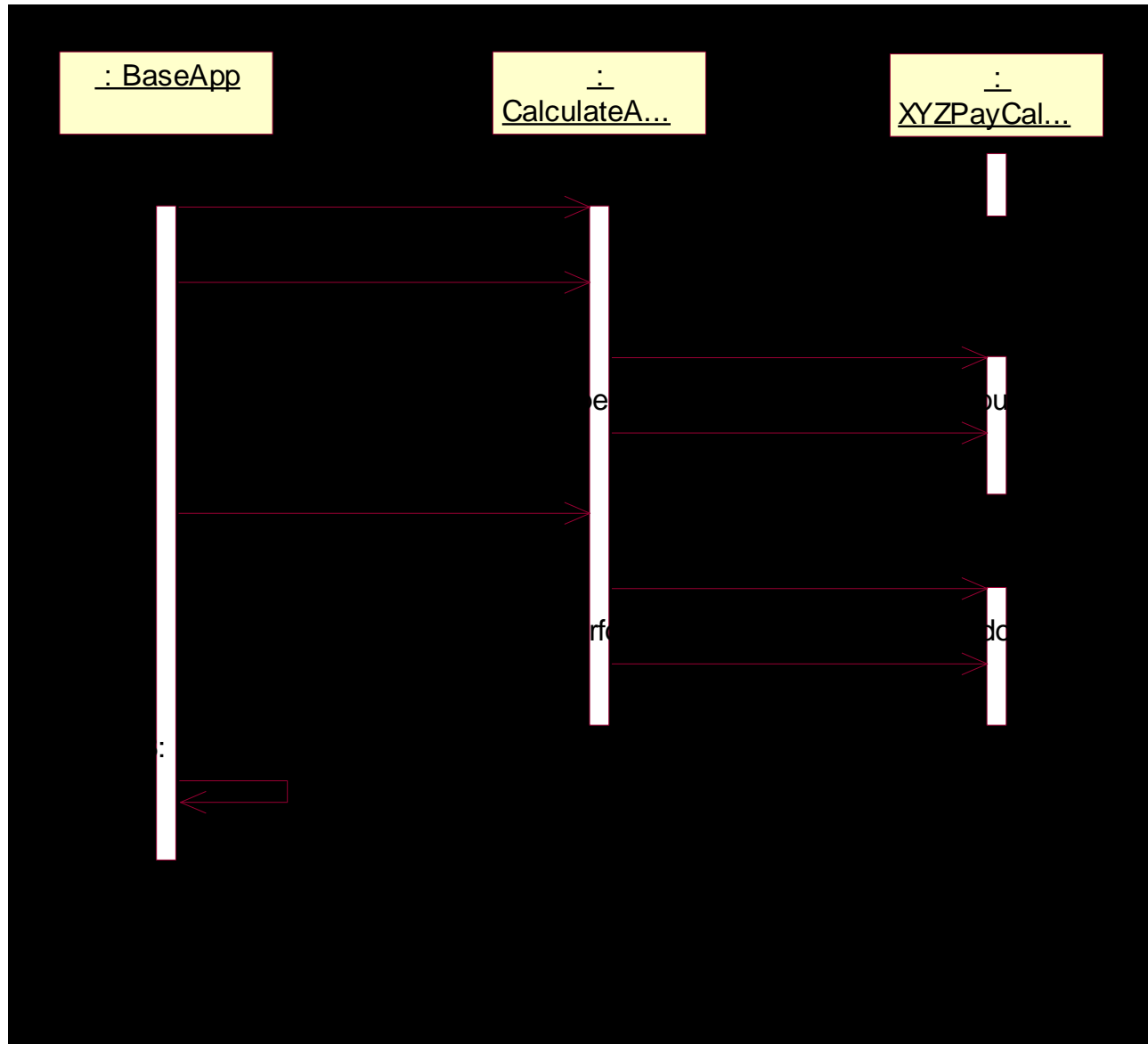
Design Classes



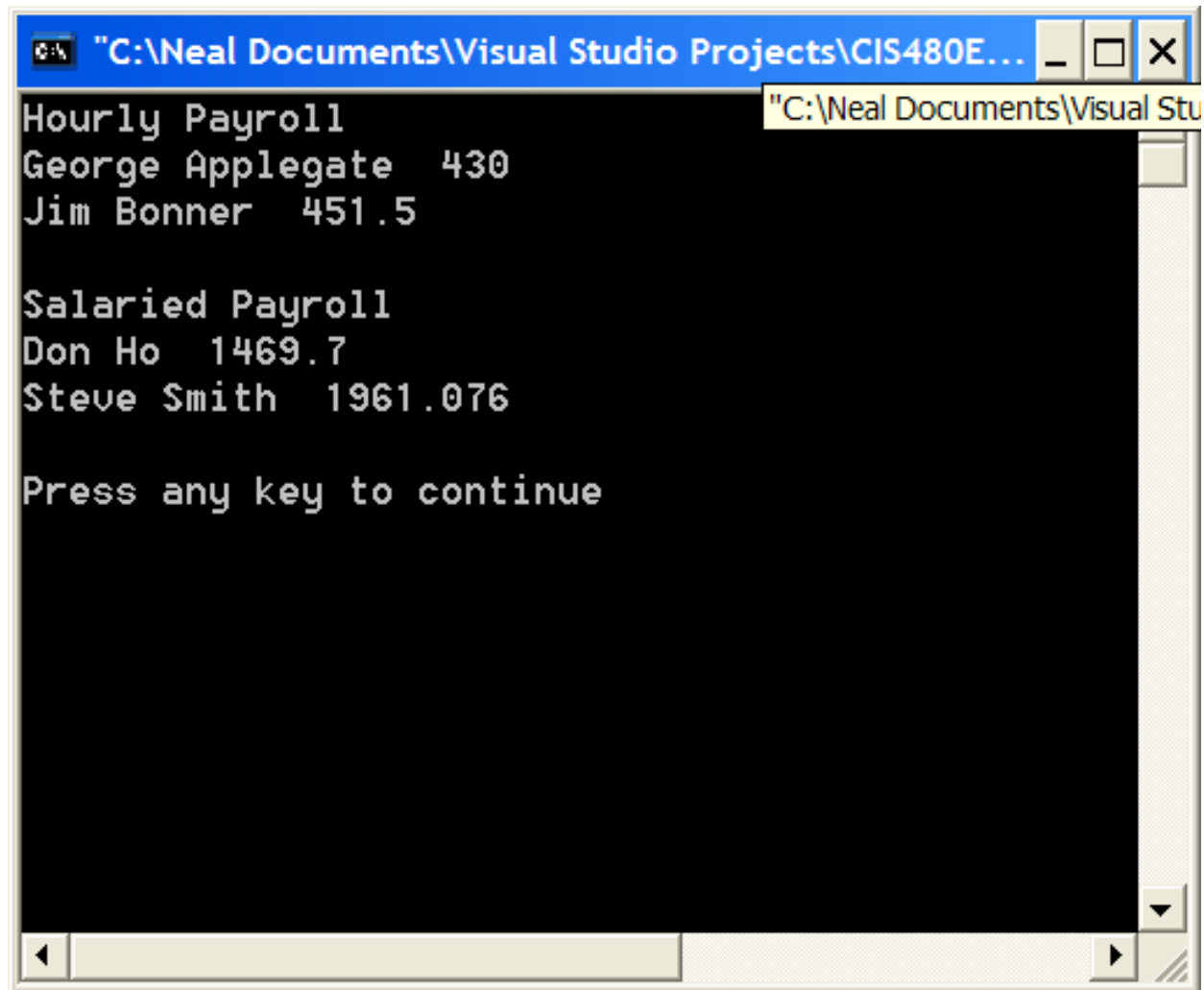
Sequence for Existing System



Sequence for Use of Adapter



Output to Console



A screenshot of a Windows console window. The title bar is blue and contains the text '"C:\Neal Documents\Visual Studio Projects\CIS480E...' followed by standard window control buttons. The console area has a black background with white text. The text displayed is as follows:

```
Hourly Payroll
George Applegate  430
Jim Bonner  451.5

Salaried Payroll
Don Ho  1469.7
Steve Smith  1961.076

Press any key to continue
```

The window includes a vertical scrollbar on the right and a horizontal scrollbar at the bottom.

namespace AdapterPattern

{

class BaseApp

{

static void Main(string[] args)

{

BaseApp ba = new BaseApp();
ba.doPayroll();

}

private void doPayroll()

{

// the only change in the application is to use the CalculateAdapter
// rather than the CalculatePay class in use currently

//

//ICalculate cp = new CalculatePay(10.75);

ICalculate cp = new CalculateAdapter();

Console.WriteLine("Hourly Payroll");

Console.WriteLine("George Applegate " + cp.calculateHourly(40.0).ToString());

Console.WriteLine("Jim Bonner " + cp.calculateHourly(42.0).ToString());

Console.WriteLine(" ");

Console.WriteLine("Salaried Payroll");

Console.WriteLine("Don Ho " + cp.calculateSalary(1, 1278.0).ToString());

Console.WriteLine("Steve Smith " + cp.calculateSalary(2, 1634.23).ToString());

Console.WriteLine(" ");

}

}

}

BaseApp Class

Only
change
one line
to
implement
XYZ

CalculatePay Class

```
namespace AdapterPattern
{
    public class CalculatePay : ICalculate
    {
        private double hourly;
        private double[] fringe;

        public CalculatePay(double h)
        {
            hourly = h;
            fringe = new double[3];
            fringe[0] = .1;
            fringe[1] = .15;
            fringe[2] = .2;
        }

        public double calculateHourly(double hours)
        {
            return (hours * hourly);
        }

        public double calculateSalary(int category, double salary)
        {
            return (salary * (1 + fringe[category]));
        }
    }
}
```

ICalculate Interface

```
namespace AdapterPattern
{
    public interface ICalculate
    {
        double calculateHourly(double hours);

        double calculateSalary(int category, double salary);
    }
}
```

CalculateAdapter Class

```
namespace AdapterPattern
{
    public class CalculateAdapter : ICalculate
    {
        public CalculateAdapter()
        {
        }

        public double calculateHourly(double hours)
        {
            XYZPayCalculator xyz = new XYZPayCalculator();
            return xyz.performHourlyPayCalculation(hours);
        }

        public double calculateSalary(int category, double salary)
        {
            XYZPayCalculator xyz = new XYZPayCalculator();
            return xyz.performSalaryiedPayCalculation(salary);
        }
    }
}
```


XYZPayCalculator Class

```
namespace AdapterPattern  
{
```

```
    public class XYZPayCalculator
```

```
    {
```

```
        private double hourly;
```

```
        public XYZPayCalculator()
```

```
        {
```

```
            hourly = 10.75;
```

```
        }
```

```
        public double performHourlyPayCalculation(double hours)
```

```
        {
```

```
            return (hours * hourly);
```

```
        }
```

```
        public double performSalaryiedPayCalculation(double salary)
```

```
        {
```

```
            double fringe;
```

```
            if (salary < 1500.00)
```

```
                fringe = .15;
```

```
            else
```

```
                fringe = .20;
```

```
            return (salary * (1 + fringe));
```

```
        }
```