# CPSC 501 – Assignment #9

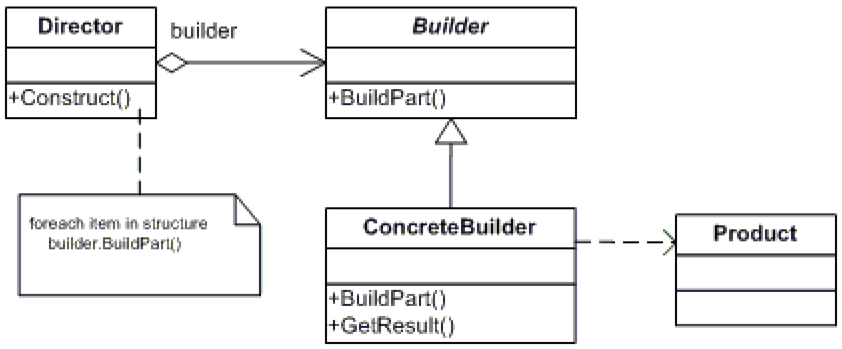
## Builder, Abstract Factory, Prototype and Visitor Patterns

Builder and Abstract Factory belong to the creational patterns category. Both of these are often used in large scale software development.

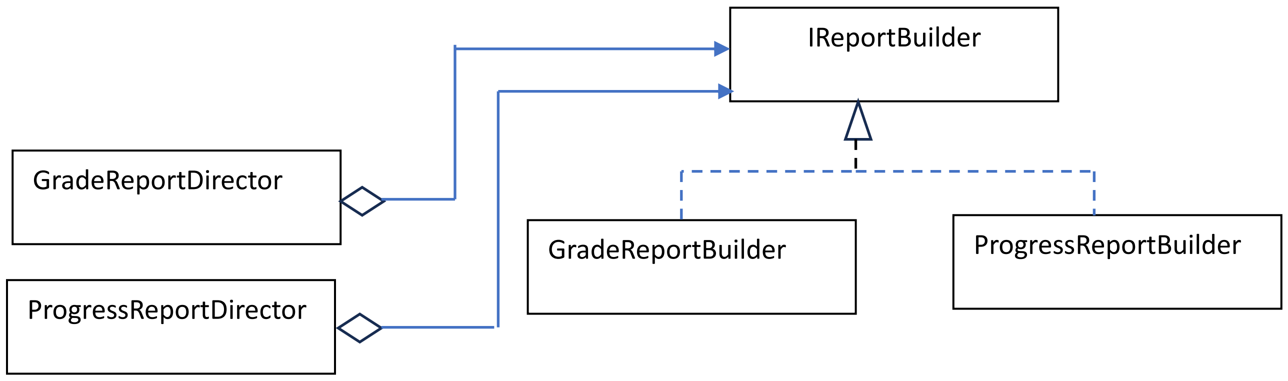
**Builder Patter**: GOF4 defines the builder pattern as:

“Separate the construction of a complex object from its representation so that the same construction process can create different representations.”. Builder pattern allows constructing a complex object step by step and returns the final software application. A good example of the use of builder pattern in creating web applications under .Net Core. The entire web application is built using the builder pattern. The pattern allows creation and assembly of the different critical parts of the web application.

The UML for the builder pattern is shown below.



The director in the above UML is responsible for assembling the entire application and also deciding on the order of the creation of the components. The builder, creates the components and usually has a method that returns the assembled final result. Note that the components needed in building an application are chosen by the director, and if different components are needed in different situations, we can use multiple directors. For example, if we want to build a student reporting application where we wanted to produce either a grade report, or a progress report, then two directors will be used as shown by the following UML.



Builder’s goal is to put together a product, so it uses an abstraction the components or parts of the products, as well as the builder itself, so that many builders can be used interchangeably, with many products (conforming to the product abstraction) and each part in the product coming from any supplier (part abstraction).

Create a windows forms application called BuilderPattern.

Add a class called Student to the project with the following code in it.

```
internal class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Major { get; set; }
    public int Test1 { get; set; }
    public int Test2 { get; set; }
    public string ComputeGrade()
    {
        string grade = "A";
        double avg = 0.4 * Test1 + 0.6 * Test2;
        if (avg > 90)
            grade = "A";
        else if (avg > 85)
            grade = "A-";
        else if (avg > 80)
            grade = "B+";
        else
            grade = "B";
        return grade;
    }
}
```

Add a class called StudentReport with the following code in it. It indicates the different components needed in building the report.

```
internal class StudentsReport
{
    public string HeaderPart { get; set; }
    public string BodyPart { get; set; } = "";
    public string ProgressPart { get; set; } = "";
    public string FooterPart { get; set; }
    public override string ToString() =>
        new StringBuilder()
        .AppendLine(HeaderPart)
        .AppendLine(BodyPart)
        .AppendLine(ProgressPart)
        .AppendLine(FooterPart)
        .ToString();
}
```

Add an interface to the project called IStudentReportBuilder with the following code in it.

```
internal interface IStudentReportBuilder
{
    void BuildHeader();
    void BuildBody();
    void BuildProgress();
    void BuildFooter();
    StudentsReport GetReport();
}
```

Add a class to the project called StudentGradeReportBuilder with the following code in it. It provides the code for building different parts of the grade report. The BuildBody code loops through the list of students it contains, and produces the computed grade for each student. The grade report does not use the progress part.

```
internal class StudentGradeReportBuilder : IStudentReportBuilder
{
    private StudentsReport _studentGradeReport;
    private List<Student> SList;
    public StudentGradeReportBuilder(List<Student> students)
    {
        SList = students;
        _studentGradeReport = new StudentsReport();
    }
    public void BuildBody()
    {
        string out1 = "";
        foreach (Student st in SList)
            out1 += $"{st.Id}\t{st.Name}\t{st.Major}\t{st.ComputeGrade()}\n";
        _studentGradeReport.BodyPart = out1;
    }


    public void BuildFooter()
    {
        _studentGradeReport.FooterPart = "\n---------Fall Semester 2023----------";
    }


    public void BuildHeader()
    {
        _studentGradeReport.HeaderPart = "\nGrade Report Date:  " +
DateTime.Now.ToShortDateString()  +"\n\n";
    }

    public void BuildProgress()
    {
        throw new NotImplementedException();
    }
    public StudentsReport GetReport() // final overall report
    {
        //Clear();
        return _studentGradeReport;
    }

    private void Clear() => _studentGradeReport = new StudentsReport();
}
```

Add a class to the project called StudentProgressReportBuilder with the following code in it. Assume that the progress report for students is generated midway through the semester where only the first test's score is available and the student has not taken the second test (i.e., the final). It provides the code for building different parts of the progress report. The BuildProgress code loops through the list of students it contains, and produces the progress comment for each student. If the first test score > 70, the progress is “Satisfactory”, otherwise the progress is “UnSatisfactory”. The progress report does not use the body part where final grades were computed in the previous grade report builder.

```
internal class StudentProgressReportBuilder:  IStudentReportBuilder
{
    private StudentsReport _studentReport;
    private List<Student> SList;
    public StudentProgressReportBuilder(List<Student> students)
    {
        SList = students;
        _studentReport = new StudentsReport();
    }
    public void BuildBody()
    {
        throw new NotImplementedException(); // no body in progress report
    }

    public void BuildFooter()
    {
        _studentReport.FooterPart = "\n---------Fall Semester 2023----------";
    }

    public void BuildHeader()
    {
        _studentReport.HeaderPart = "\nProgress Report Date: " +
DateTime.Now.ToShortDateString() + "\n\n";
    }

    public void BuildProgress()
    {
        string out1 = "";
        foreach (Student st in SList)
            if (st.Test1 > 70)
                out1 += $"{st.Id}\t{st.Name}\t{st.Major}\t{st.Test1} - Satisfactory\n";
            else
                out1 += $"{st.Id}\t{st.Name}\t{st.Major}\t{st.Test1} - UnSatisfactory\n";
        _studentReport.BodyPart = out1;
    }
    public StudentsReport GetReport()
    {
        //Clear();
        return _studentReport;
    }

    private void Clear() => _studentReport = new StudentsReport();
}
```

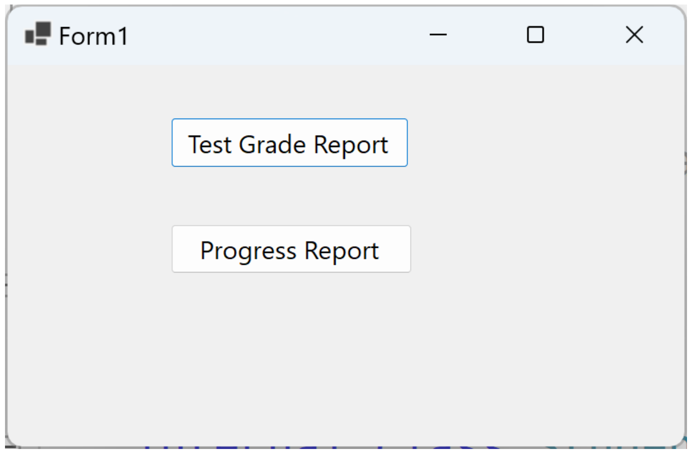
Add a class to the project called StudentGradeReportDirector with the following code in it. It provides injection of the builder in the constructor and then calls on the different methods of the builder to build the grade report.

```
internal class StudentGradeReportDirector
{
    private readonly IStudentReportBuilder _studentGradeReportBuilder;
    public StudentGradeReportDirector(IStudentReportBuilder studentGradeReportBuilder)
    {
        _studentGradeReportBuilder = studentGradeReportBuilder;
    }

    public void BuildStudentsGradeReport() // order in which components will be built
    {
        _studentGradeReportBuilder.BuildHeader();
        _studentGradeReportBuilder.BuildBody();
        _studentGradeReportBuilder.BuildFooter();
    }
}
```

Add a class to the project called StudentProgressReportDirector with the following code in it. It provides injection of the builder in the constructor and then calls on the different methods of the builder to build the progress report.

```
internal class StudentProgressReportDirector
{
    private readonly IStudentReportBuilder _studentProgressReportBuilder;
    public StudentProgressReportDirector(IStudentReportBuilder 
studentProgressReportBuilder)
    {
        _studentProgressReportBuilder = studentProgressReportBuilder;
    }

    public void BuildStudentsProgressReport()
    {
        _studentProgressReportBuilder.BuildHeader();
        _studentProgressReportBuilder.BuildProgress();
        _studentProgressReportBuilder.BuildFooter();
    }
}
```

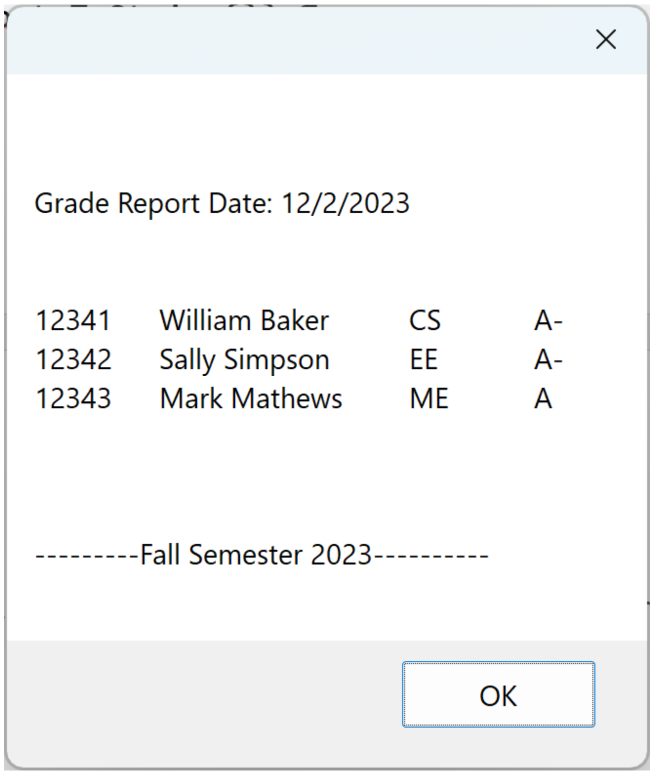
Put two buttons in the form with names of “btnTestGradeReport” and “btnTestProgressReport” as shown below.



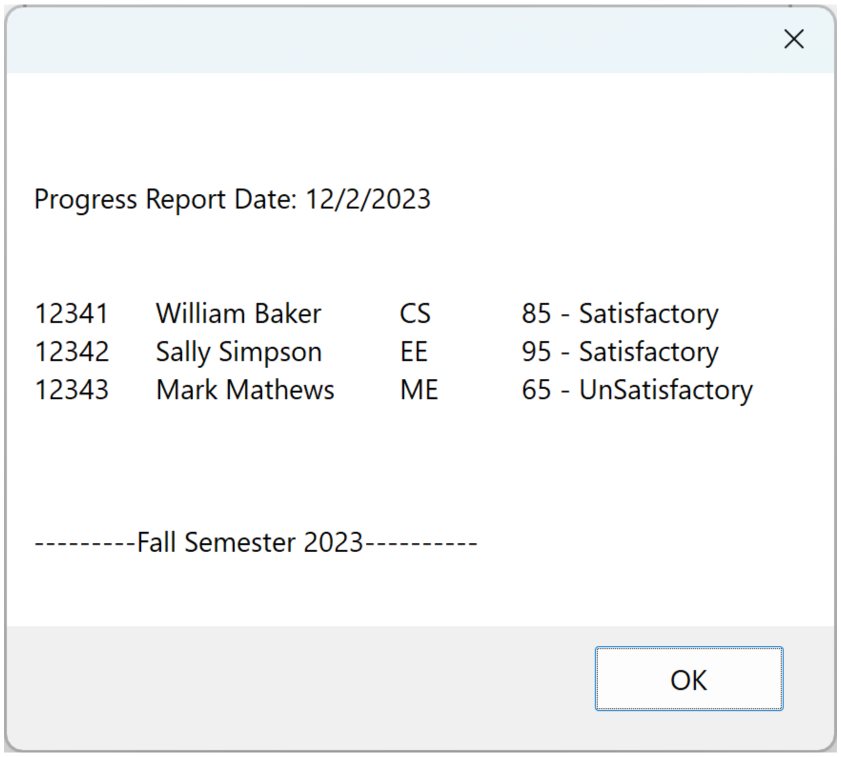
Double click on each button to write the event handlers. The code for the two button handlers is shown below.

```
private void btnTestGradeReport_Click(object sender, EventArgs e)
{
    var SList = new List<Student>
    {
        new Student { Id= 12341, Major="CS", Name="William Baker", Test1=85,
Test2=91},
        new Student { Id= 12342, Major="EE", Name="Sally Simpson", Test1=81,
Test2=88},
        new Student { Id= 12343, Major="ME", Name="Mark Mathews", Test1=89,
Test2=95},
    };
    var builder = new StudentGradeReportBuilder(SList);
    var director = new StudentGradeReportDirector(builder);
    director.BuildStudentsGradeReport();
    var report = builder.GetReport();
    MessageBox.Show(report.ToString());
}

private void btnTestProgressReport_Click(object sender, EventArgs e)
{
    var SList = new List<Student>
    {
        new Student { Id= 12341, Major="CS", Name="William Baker", Test1=85},
        new Student { Id= 12342, Major="EE", Name="Sally Simpson", Test1=95},
        new Student { Id= 12343, Major="ME", Name="Mark Mathews", Test1=65},
    };
    var builder = new StudentProgressReportBuilder(SList);
    var director = new StudentProgressReportDirector(builder);
    director.BuildStudentsProgressReport();
    var report = builder.GetReport();
    MessageBox.Show(report.ToString());
}
```

Run the project. Once you click on each of the two buttons, the grade report and the progress report are displayed as:



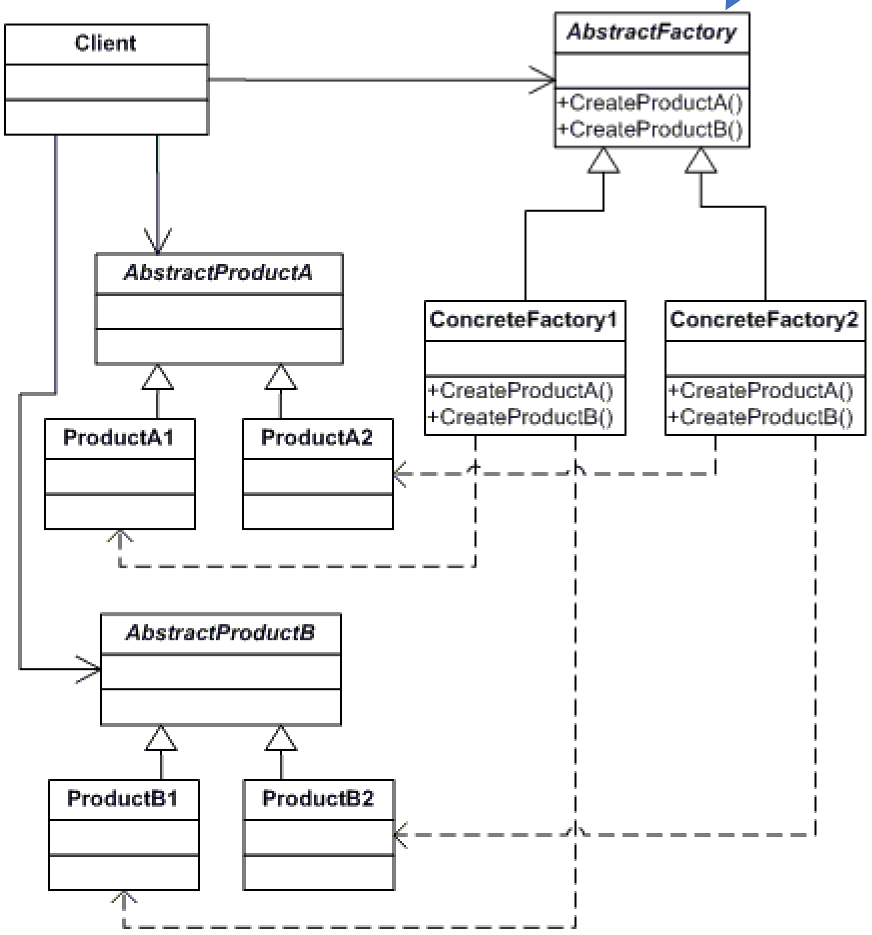
```
Grade Report Date: 12/2/2023

12341    William Baker     CS      A-
12342    Sally Simpson     EE      A-
12343    Mark Mathews      ME      A

---------Fall Semester 2023----------
```



```
Progress Report Date: 12/2/2023

12341    William Baker     CS      85 - Satisfactory
12342    Sally Simpson     EE      95 - Satisfactory
12343    Mark Mathews      ME      65 - UnSatisfactory

---------Fall Semester 2023----------
```

## Abstract Factory Pattern:

Abstract Factory pattern is perhaps the most difficult pattern to understand. As you can see from the following UML diagram, it has quite a few interactions of different classes and inheritance from abstract classes. The goal of the abstract factory pattern is to enforce a set of interfaces to a more involved workflow where many classes will be needed for implementing each part of the workflow. Further, the actual code in the methods may be different so we need to isolate the creation of the class that is part of the work flow (similar to how a regular factory pattern, where the user does not exactly specifies the name of class to be created. The abstract factory pattern starts out by an abstract class where the abstract methods return an interface. Each method indicates which type of class is to be created (not the exact name of class but which interface the created class implements).



For example, in creating a LoanProcessing application, the starting point will be the abstract class:

```
abstract class LoanFactory
{
    public abstract ICreditCheck CreateCreditCheck();
    public abstract IBalanceCheck CreateBalanceCheck();
}
```

Each method in the above class will return an object of a class that implements ICreditCheck, or IBalanceCheck.

For example, the ICreditCheck may appear as:

```
interface ICreditCheck
{
    int GetCreditScore(string firstName, string lastName);
}
```

If we had branches for our bank in US and Canada, then the GetCreditScore implementation will be different as the credit rating scores are different in each country. This is where a separate class will be created for checking the credit for eah country, but both classes will implement the ICreditCheck to provide a uniformity in creating the loan application for different countries. Creation of such classes with different implementations is integrated in an enhanced factory patter referred to as the abstract factory pattern. Lets put more detail in understanding this pattern by creating a simple Loan Processing application for US and Canada.

Create a new windows forms project called AbstractFactoryPattern.
Add a class called BankInfo to the project with the following code:

```
class BankInfo
{
    string firstName;
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    string lastName;
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }

    string bankName;
    public string BankName
    {
        get { return bankName; }
        set { bankName = value; }
    }

    long checkingAccountNum;
    public long CheckingAccountNum
    {
        get { return checkingAccountNum; }
        set { checkingAccountNum = value; }
    }

    long savingAccountNum;
    public long SavingAccountNum
    {
        get { return savingAccountNum; }
        set { savingAccountNum = value; }
    }
}
```

Add an interface to the project called ICreditCheck with the following code.

```
interface ICreditCheck
{
    int GetCreditScore(string firstName, string lastName);
}
```

Add an interface called IBalanceCheck with the following code in it.

```
interface IBalanceCheck
{
    double GetCurrentBalances(List<BankInfo> BInfoList);
    double GetCurrentLoans(List<BankInfo> BInfoList);
}
```

Add a class called LoanFactory (abstract factory class) with the following code in it.

```
abstract class LoanFactory
{
    public abstract ICreditCheck CreateCreditCheck();
    public abstract IBalanceCheck CreateBalanceCheck();
}
```

The above abstract factory class acts as the base class for all concrete factory classes. For example, if we wanted to provide loan services in US and Canada, we will provide two concrete factories, one for each country. Add a class called LoanFactoryUS with the following code in it. As you can see, it inherits from abstract base factory class and overrides the methods that create concrete classes for a particular country.

```
class LoanFactoryUS : LoanFactory  // concrete factory
{
    public override ICreditCheck CreateCreditCheck()
    {
        return new CreditCheckUS();
    }

    public override IBalanceCheck CreateBalanceCheck()
    {
        return new BalanceCheckUS();
    }
}
```

Add a class called CreditCheckUS with the following code in it.

```
class CreditCheckUS : ICreditCheck
{
    public int GetCreditScore(string firstName, string lastName)
    {
        // call Equifax, Transunion etc.. services
        int score = 0;
        if ((firstName.ToUpper() == "BILL") && (lastName.ToUpper() == "BAKER"))
            score = 630;
        if ((firstName.ToUpper() == "SALLY") && (lastName.ToUpper() == "SIMPSON"))
            score = 720;
        return score;
    }
}
```

Add a class called BalanceCheckUS with the following code in it.

```
class BalanceCheckUS : IBalanceCheck
{
    public double GetCurrentBalances(List<BankInfo> BInfoList)
    {
        // call each banks service to obtain info
        double bal = 0;
        if (BInfoList.Count > 0)
        {
            if ((BInfoList[0].FirstName.ToUpper() == "BILL") &&
                (BInfoList[0].LastName.ToUpper() == "BAKER"))
                bal = 5728;
            if ((BInfoList[0].FirstName.ToUpper() == "SALLY") &&
                (BInfoList[0].LastName.ToUpper() == "SIMPSON"))
                bal = 14455;
        }
        return bal;
    }

    public double GetCurrentLoans(List<BankInfo> BInfoList)
    {
        // call each banks service to obtain info
        double loanamt = 0;
        if (BInfoList.Count > 0)
        {
            if ((BInfoList[0].FirstName.ToUpper() == "BILL") &&
                (BInfoList[0].LastName.ToUpper() == "BAKER"))
                loanamt = 10000;
            if ((BInfoList[0].FirstName.ToUpper() == "SALLY") &&
                (BInfoList[0].LastName.ToUpper() == "SIMPSON"))
                loanamt = 5000;
        }
        return loanamt;
    }
}
```

Similarly to provide loan services for Canada, create a class called LoanFactoryCanada which inherits from LoanFactory (abstract factory class) as shown below.

```
class LoanFactoryCanada : LoanFactory // concrete factory
{
    public override ICreditCheck CreateCreditCheck()
    {
        return new CreditCheckCanada();
    }

    public override IBalanceCheck CreateBalanceCheck()
    {
        return new BalanceCheckCanada();
    }
}
```

Add a class called CreditCheckCanada with the following code in it.

```
class CreditCheckCanada : ICreditCheck
{
    public int GetCreditScore(string firstName, string lastName)
    {
        // call Trans Union or other Canadian credit check services
```

```
            // A Canadian company's CREDIT INFO SCORE can range from 0-70,
details as follows;
            //C.I. Range: Risk Factor
            //0 - 11 Minimal
            //12 - 23 Average
            //24 - 35 Marginal
            //36 - 47 High
            //48 - 59 Very High
            //60 - 70 Poor
            int score = 0;
            if ((firstName.ToUpper() == "MARK") && (lastName.ToUpper() ==
"MATHEWS"))
                score = 32;
            if ((firstName.ToUpper() == "NANCY") && (lastName.ToUpper() ==
"ADAMS"))
                score = 8;
            return score;
        }
    }
```

Add a class called BalanceCheckCanada with the following code in it.

```
    class BalanceCheckCanada : IBalanceCheck
    {
        public double GetCurrentBalances(List<BankInfo> BInfoList)
        {
            // call each banks service to obtain info
            double bal = 0;
            if (BInfoList.Count > 0)
            {
                if ((BInfoList[0].FirstName.ToUpper() == "MARK") &&
                    (BInfoList[0].LastName.ToUpper() == "MATHEWS"))
                    bal = 6520;
                if ((BInfoList[0].FirstName.ToUpper() == "NANCY") &&
                    (BInfoList[0].LastName.ToUpper() == "ADAMS"))
                    bal = 5875;
            }
            return bal;
        }

        public double GetCurrentLoans(List<BankInfo> BInfoList)
        {
            // call each banks service to obtain info
            double loanamt = 0;
            if (BInfoList.Count > 0)
            {
                if ((BInfoList[0].FirstName.ToUpper() == "NANCY") &&
                    (BInfoList[0].LastName.ToUpper() == "ADAMS"))
                    loanamt = 200;
                if ((BInfoList[0].FirstName.ToUpper() == "MARK") &&
                    (BInfoList[0].LastName.ToUpper() == "MATHEWS"))
                    loanamt = 7000;
            }
            return loanamt;
        }
    }
```

To test the abstract factory pattern, add two buttons to the form with a names of btnAbstractFactoryUS and  btnAbstractFactoryCanada with the following codes in their handlers.

```
private void btnAbstractFactoryUS_Click(object sender, EventArgs e)
        {
            LoanFactory lf = new LoanFactoryUS();
            ICreditCheck icc = lf.CreateCreditCheck();
            IBalanceCheck ibc = lf.CreateBalanceCheck();
            BankInfo bi = new BankInfo();
            bi.BankName = "City Bank";
            bi.CheckingAccountNum = 1234;
            bi.SavingAccountNum = 12341;
            bi.FirstName = "Bill";
            bi.LastName = "Baker";
            List<BankInfo> BList = new List<BankInfo>();
            BList.Add(bi);

            double creditScore = icc.GetCreditScore("Bill", "baker");
            double balance = ibc.GetCurrentBalances(BList);
            MessageBox.Show("Credit Score for Bill = " + creditScore.ToString() +
                "\nBalance = " + balance.ToString());
      }

private void btnAbstractFactoryCanada_Click(object sender, EventArgs e)
        {
            LoanFactory lf2 = new LoanFactoryCanada();
            ICreditCheck icc2 = lf2.CreateCreditCheck();
            IBalanceCheck ibc2 = lf2.CreateBalanceCheck();
            BankInfo bi2 = new BankInfo();
            bi2.BankName = "Bank of Canada";
            bi2.CheckingAccountNum = 66554;
            bi2.SavingAccountNum = 86865;
            bi2.FirstName = "Nancy";
            bi2.LastName = "Adams";
            List<BankInfo> BList2 = new List<BankInfo>();
            BList2.Add(bi2);

            double creditScore2 = icc2.GetCreditScore("Nancy", "Adams");
            double balance2 = ibc2.GetCurrentBalances(BList2);
            MessageBox.Show("Credit Score for Nancy (Canada) = " +
creditScore2.ToString() +
                "\nBalance = " + balance2.ToString());
        }
```

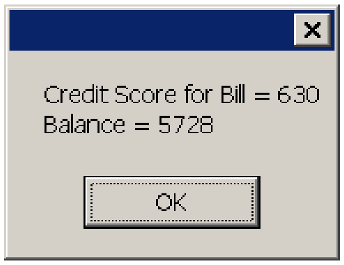
Running the program and clicking on the above buttons produces the following output. Note that the difference in calling the Loan info for the US and Canada is just the first line of code i.e., for US, is is:

```
            LoanFactory lf = new LoanFactoryUS();
```

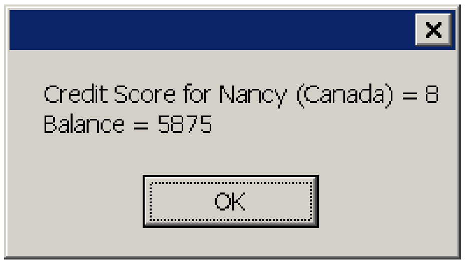
And for Canada, it is:

```
            LoanFactory lf2 = new LoanFactoryCanada();
```

Rest of the code for determining the approval of loan i.e., checking the credits and the bank balances is exactly the same for the two countries. This is the beauty of the abstract factory pattern.



```
Credit Score for Bill = 630
Balance = 5728
```



```
Credit Score for Nancy (Canada) = 8
Balance = 5875
```

## Visitor Patter:

Visitor pattern as proposed by GoF4 is somewhat complex. C# extension methods provide a lot cleaner solution to the Visitor pattern with the goal of adding a new method to an existing class for which we do not have the source code.. To show this, suppose we have an employee class with the following code in it.

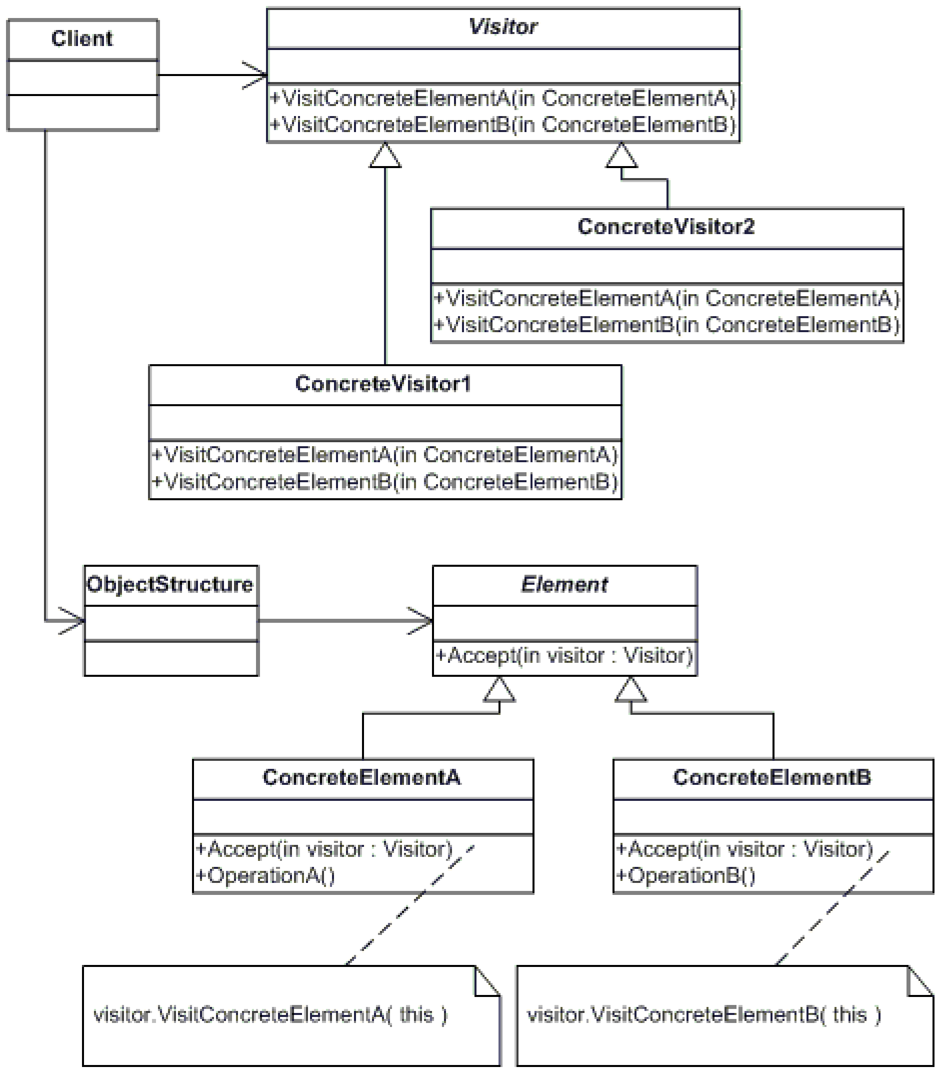
```
internal class Employee
{
    public string Name { get; set; }
    public int Id { get; set; }
    public double HoursWorked { get; set; }
    public double PayRate { get; set; }
    public double ComputePay()
    {
        return PayRate * HoursWorked;
    }
}
```

If we wanted to add a method called ComputeOverTimePay, we can create another class called MyExtension (the name of the class is not important). The rule for extension methods in C# is that the extending class should be static (meaning all methods in it to be static), The method we want to add to the existing class must indicate by the first parameter as to which class are we extending.

```
static class MyExtension  // all functions inside have to be static
{
    public static double ComputeOvertimePay(this Employee emp, double 
overtimeRate)
    {
        double overtimePay = 0;
        if (emp.HoursWorked > 40)
        {
            var overtimeHours = emp.HoursWorked - 40;
            overtimePay = overtimeHours * overtimeRate * emp.PayRate;
        }
        return overtimePay;
    }
}
```

Now the employee class will behave as if it contains the ComputeOvertimePay method.

The GoF4 Visitor pattern is also useful in collecting information from a collection of unrelated classes. The UML for the visitor pattern is shown below. Read the following design lightly to get a general idea of the GoF4 Visitor pattern. As mentioned earlier, if you are using C#, then the extension method is a cleaner solution. In Java, C++, or Python, the following approach for the Visitor pattern will be needed.


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA( this )
visitor.VisitConcreteElementB( this )

<u>Optional Part for Assignment</u>: Create a new windows forms project called VisitorPattern. Add an interface called IVisitor with the following code in it.

```
interface IVisitor
{
    void Visit(Element element);
}
```

Add a class called Element that will act as the base class for classes that will allow visitors.

```
abstract class Element
{
    public abstract void Accept(IVisitor visitor);
}
```

Add a base class called Employee for different types of employees with the following code in it.

```
class Employee : Element
{
    public Employee(string nm, int vacDays, double bonus)
    {
        name = nm;
        vacationDays = vacDays;
        this.bonus = bonus;
    }

    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    int vacationDays;
    public int VacationDays
    {
        get { return vacationDays; }
        set { vacationDays = value; }
    }

    double bonus;
    public double Bonus
    {
        get { return bonus; }
        set { bonus = value; }
    }

    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }

    public override string ToString()
    {
        return name + " : " + vacationDays.ToString() + " : " +
            bonus.ToString();
    }
}
```

Add three classes for three types of employees as shown below.

```
class AdminAssistant : Employee
{
    public AdminAssistant(string nm, int vacDays, double pay) :
        base(nm, vacDays, pay)
    {
    }
}

class Manager : Employee
{
    public Manager(string nm, int vacDays, double pay) :
        base(nm, vacDays, pay)
    {
    }
}
```

```
class VicePresident : Employee
{
    public VicePresident(string nm, int vacDays, double pay) :
        base(nm, vacDays, pay)
    {
    }
}
```

Add a class called Employees that maintains a list of employee objects, with the code as shown below.

```
class Employees
{
    List<Employee> _EList = new List<Employee>();
    internal List<Employee> EList
    {
        get { return _EList; }
    }

    public void Add(Employee emp)
    {
        _EList.Add(emp);
    }

    public void Remove(Employee emp)
    {
        _EList.Remove(emp);
    }

    public void Accept(IVisitor visitor)
    {
        foreach (Employee em in _EList)
        {
            em.Accept(visitor);
        }
    }
}
```

Add a class called BonusVisitor which implements a concrete visitor with the following code in it. As you can see, it updates the bonus value of an employee by 10 %.

```
class BonusVisitor : IVisitor
{
    #region IVisitor Members
    public void Visit(Element element)
    {
        Employee emp = element as Employee;
        emp.Bonus = emp.Bonus * 1.10;
    }
    #endregion
}
```

Add another conceret visitor class called VacationVisitor that increases the number of vacation days for an employee by 2.

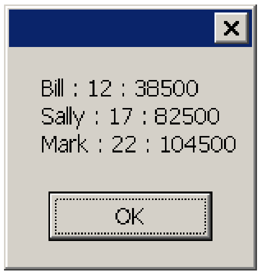
```
class VacationVisitor : IVisitor
{
    #region IVisitor Members

    public void Visit(Element element)
    {
        Employee e1 = element as Employee;
```

```
            e1.VacationDays += 2;
        }

        #endregion
    }
```

To test the visitor pattern, add a button to the form called btnVisitor with the following code in it.

```
private void btnVisitor_Click(object sender, EventArgs e)
        {
            Employees emps = new Employees();
            AdminAssistant aa = new AdminAssistant("Bill",10,35000);
            emps.Add(aa);
            Manager mn = new Manager("Sally", 15, 75000);
            emps.Add(mn);
            VicePresident vp = new VicePresident("Mark", 20, 95000);
            emps.Add(vp);

            BonusVisitor ivis = new BonusVisitor();
            emps.Accept(ivis);

            VacationVisitor vv = new VacationVisitor();
            emps.Accept(vv);

            string out1 = "";
            foreach (Employee ee in emps.EList)
                out1 += ee.ToString() + "\n";
            MessageBox.Show(out1);
        }
```

If you run the program and click on the above button, you will see the following output.



```
Bill : 12 : 38500
Sally : 17 : 82500
Mark : 22 : 104500
```

## Using Visitor to Add New Methods to an Existing Class:

To demonstrate the capability that visitor can add new functionality to a class without having its source code being modified, add a generic interface called IVisitor2 with the following code in it.

```
    interface IVisitor2<T>
    {
        T Visit (Element2 element);
    }
```

Add a class called Element2 with the following code in it.

```
    abstract class Element2
    {
        public abstract T Accept<T>(IVisitor2<T> visitor);
    }
```

Add a class called Employee2 that inherits from the Element2 class as shown below.

```
class Employee2 : Element2
{
    public Employee2(string nm, int hworked, double rate)
    {
        name = nm;
        hoursWorked = hworked;
        this.payRate = rate;
    }

    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    int hoursWorked;
    public int HoursWorked
    {
        get { return hoursWorked; }
        set { hoursWorked = value; }
    }

    double payRate;
    public double PayRate
    {
        get { return payRate; }
        set { payRate = value; }
    }


    public override T Accept<T>(IVisitor2<T> visitor)
    {
        return visitor.Visit(this);
    }

    public override string ToString()
    {
        return name + " : " + hoursWorked.ToString() + " : " +
            payRate.ToString();
    }
}
```

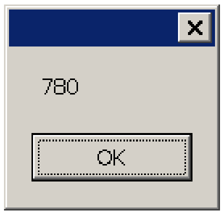
Add a conceret visitor class called ComputePayVisitor with the following code in it.

```
class ComputePayVisitor:IVisitor2<double>
{
    #region IVisitor2 Members

    public double Visit(Element2 element)
    {
        Employee2 emp2 = element as Employee2;
        return (emp2.HoursWorked * emp2.PayRate);
    }

    #endregion
}
```

To test the above visitor, add a button called btnAddComputePayMethod with the following code in it.

```
private void btnAddComputePayMethod_Click(object sender, EventArgs e)
        {
            Employee2 em = new Employee2("John Jacobs", 52, 15);
            ComputePayVisitor cpv = new ComputePayVisitor();
            double res = em.Accept(cpv);
            MessageBox.Show(res.ToString());
        }
```

If you run the program and click on the above button, you will get the following output.



```
780
```

Exercise:  Add another concrete visitor called OverTimeVisitor.

Solution:

```
    class OverTimeVisitor : IVisitor2<float>
    {
        #region IVisitor2<float> Members

        public float Visit(Element2 element)
        {
            Employee2 emp2 = element as Employee2;
            float overTime = 0;
            if (emp2.HoursWorked > 40)
                overTime = (float)((emp2.HoursWorked - 40) * emp2.PayRate *
1.5);
            return overTime;
        }

        #endregion
    }
```

Test Code for OverTime Visitor:

```
private void btnAddOverTime_Click(object sender, EventArgs e)
        {
            Employee2 em = new Employee2("John Jacobs", 52, 15);
            OverTimeVisitor otv = new OverTimeVisitor();
            double res = em.Accept(otv);
            MessageBox.Show("OverTime Pay = " + res.ToString());
        }
```

The disadvantage of the Visitor pattern is that the functionality that it provides requires each class to provide a visitor method, and thus the design becomes a little involved. Since, we can easily use a foreach loop to calculate aggregates or do updates e.g., the bonus for employees can be updated as:

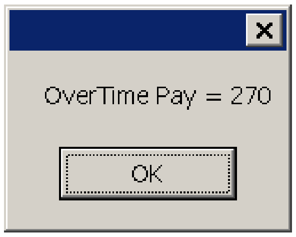
```
            foreach (Employee ee in emps.EList)
                ee.Bonus = ee.Bonus * 1.1;
```

Similarly, to add a new method to an existing class, the C# extension methods provide a lot cleaner solution. To show this, add a class called OverTimeExtension with the following code in it.

```
    static class OverTimeExtension
    {
        public static double ComputeOverTime(this Employee2 e2)
        {
            double overTimePay = 0;
            if (e2.HoursWorked > 40)
            {
                overTimePay = (e2.HoursWorked - 40) * 1.5 * e2.PayRate;
            }
            return overTimePay;
        }
    }
```

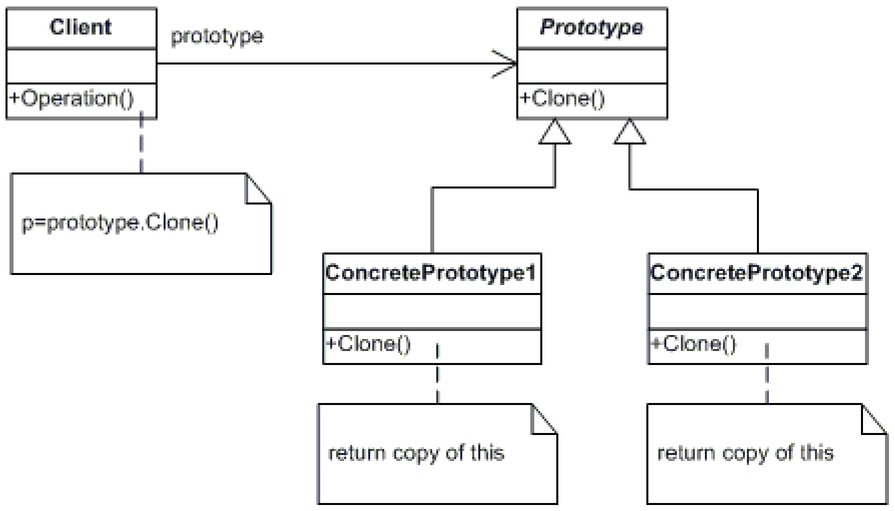
Add a button called btnOverTimeExtension with the following code in it.

```
private void btnOverTimeExtension_Click(object sender, EventArgs e)
        {
            Employee2 e2 = new Employee2("John Jacobs", 52, 15);
            double res = e2.ComputeOverTime();
            MessageBox.Show("OverTime Pay = " + res.ToString());
        }
```


OverTime Pay = 270

## Prototype Pattern:

The purpose of prototype pattern is to create clone of an existing object i.e., in memory copy of an existing object. C# and Java provide a cleaner solution to this pattern via the ICloneable (or Cloneable in Java) interface. You had implemented the ICloneable in one of your previous assignments. This may be needed in an algorithm implementation e.g., a Genetic algorithm requires cloning of a population member from one generation to another. Another reason for cloning is to efficiently create an object which is already initialized to a given state. The UML for the Prototype pattern according to GoF4 is shown below.



Optional Part for Assignment:: Create a windows forms application project called PrototypePattern. Add a class called PrototypeBase with the following code in it.

```
[Serializable]
abstract class ProtoTypeBase<T>
{
    public T Copy()
    {
        MemoryStream mstr = new MemoryStream();
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(mstr, this);
        mstr.Seek(0, SeekOrigin.Begin);
        T cp = (T) bf.Deserialize(mstr);
        return cp;
    }
}
```

Add a class called Employee with the following code in it.

```
[Serializable]
class Employee : ProtoTypeBase<Employee>
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Branch { get; set; }
    public int EmployeeId { get; set; }
    public Address Addr { get; set; }
}
```

Add a class called Address with the following code in it.

```
    [Serializable]
    class Address
    {
        public string StreetAddress { get; set; }
        public string City { get; set; }
    }
```

Add a class called EmployeeCopyManager with the following code. The copy manager returns a copy of an employee with the default fields already populated.

```
    class EmployeeCopyManager : ProtoTypeBase<Employee>
    {
        Dictionary<String, Employee> DTable = new Dictionary<String,
Employee>();

        public EmployeeCopyManager()
        {
            Address a1 = new Address
            {
                StreetAddress = "55 Pizza Lane",
                City = "Austin"
            };
            Employee e1 = new Employee();
            e1.FirstName = "";
            e1.LastName = "";
            e1.EmployeeId = 0;
            e1.Branch = "Austin";
            e1.Addr = a1;
            DTable.Add(e1.Branch, e1);
        }

        public Employee this[string bran]
        {
            get
            {
                return (DTable[bran]).Copy();
            }
        }
    }
```

To test the prototype pattern, add a button to the form called btnProtoType with the following code in it.

```
        private void btnProtoType_Click(object sender, EventArgs e)
        {
            Address a1 = new Address
            {
                StreetAddress = "55 Pizza Lane",
                City = "Austin"
            };
            Employee e1 = new Employee();
            e1.FirstName = "Bill";
            e1.LastName = "Baker";
            e1.EmployeeId = 12345;
            e1.Addr = a1;
            Employee e2 = e1.Copy();
            e1.Addr.StreetAddress = "25 Taco Lane";
            MessageBox.Show(e2.Addr.StreetAddress);
        }
```

Add a button named btnTestPrototypeManage with the following code in the handler.

```
private void btnPrototypeManager_Click(object sender, EventArgs e)
   {
       EmployeeCopyManager emg = new EmployeeCopyManager();
       Employee e1 = emg["Austin"];
       MessageBox.Show(e1.Addr.StreetAddress);
   }
```