https://dotnetcodr.com/architecture-and-patterns/

Adapter Design Pattern

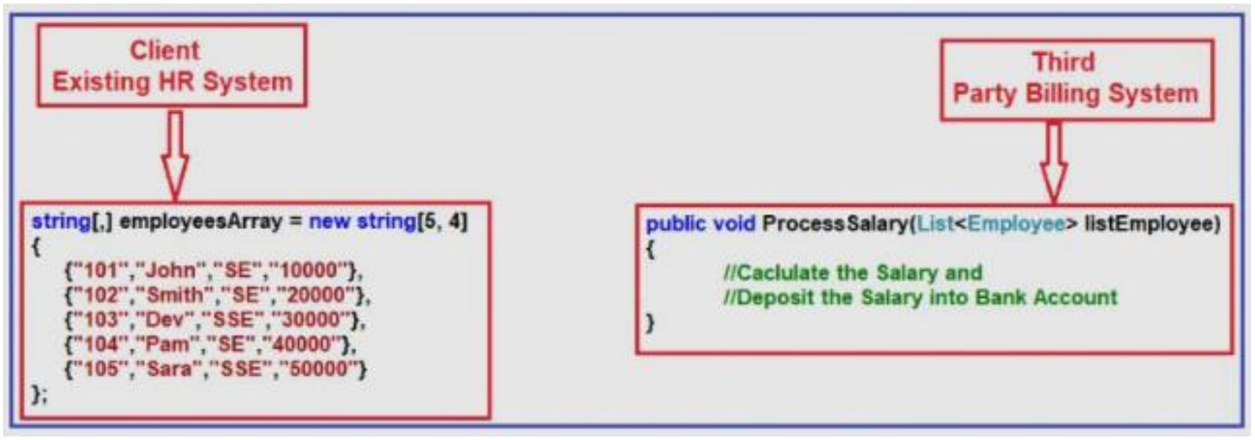
The Adapter Design Pattern is a Structural Design Pattern that allows incompatible interfaces (objects) to work together. The Adapter Design Pattern acts as a bridge between two incompatible objects. Let's say the first object is A and the second object is B. And object A wants to consume some of the services provided by object B. As they are incompatible so they cannot communicate directly.

In this case, Adapter will come into the picture and will act as a middleman or bridge between object A and object B. Now, object A will call the Adapter and Adapter will do the necessary transformations or conversions and then it will call object B

The Adapter Design Pattern in C# involves a single class called Adapter which is responsible for communication between two independent or incompatible interfaces. So, in simple words, we can say that the Adapter Design Pattern helps two incompatible interfaces to work together. We can also say that it works like a Wrapper which makes two incompatible systems work together.

Example to Understand Adapter Design Pattern in C#:

Here, you can see two interfaces or you can say two systems. On the right-hand side, you can see the Third Party Billing System and on the left-hand side, you can see the Client i.e. the Existing HR System. Now, we will see how these two systems are incompatible and we will also see how we will make them compatible using Adapter Design Patterns in C#.



```
string[,] employeesArray = new string[5, 4]
{
    {"101","John","SE","10000"},
    {"102","Smith","SE","20000"},
    {"103","Dev","SSE","30000"},
    {"104","Pam","SE","40000"},
    {"105","Sara","SSE","50000"}
};
```

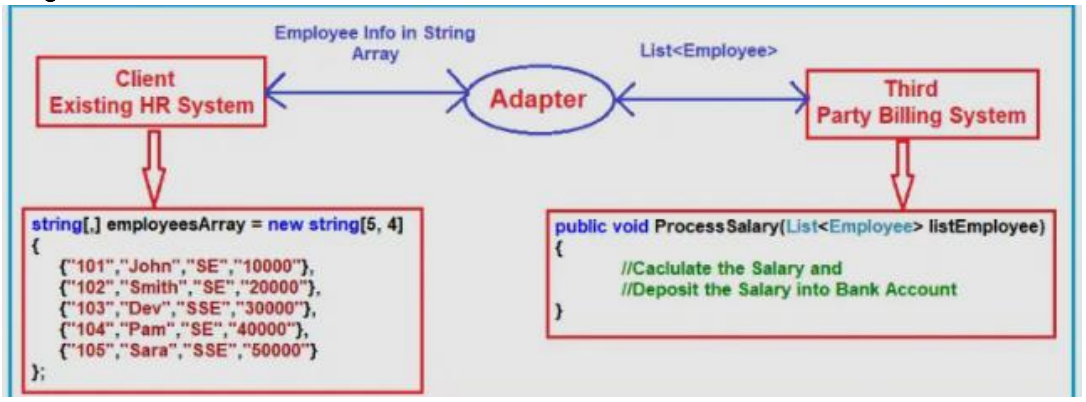
```
public void ProcessSalary(List<Employee> listEmployee)
{
        //Caclulate the Salary and
        //Deposit the Salary into Bank Account
}
```

As you can see, the Third Party Billing System provides one functionality called ProcessSalary. What this ProcessSalary method will do is, it will take the employee list (i.e. **List<Employee>**) as an input parameter and then loop through each employee and calculate the salary and deposit the salary into the employee's bank account.

On the left-hand side i.e. in the Existing HR System, the employee information is stored in the form of a string array. The HR System wants to process the salary of employees. Then what the HR System has to do is, it has to call the ProcessSalary method of the Third Party Billing System. But if you look at the HR system, the employee information is stored in the form of a string array and the ProcessSalary method of the Third Party Billing System wants data in List<Employee>. So, the HR System cannot call directly to the Third Party Billing System because List<Employee> and string array are not compatible. So, these two systems are incompatible.

How we can make these two incompatible systems work together?

We can use the Adapter Design Pattern in C# to make these two systems or interfaces work together. Now, we need to introduce an Adapter between the HR System and the Third Party Billing System as shown in the below image


Employee Info in String Array
List<Employee>
Client
Existing HR System
Adapter
Third
Party Billing System
//Caclulate the Salary and
//Deposit the Salary into Bank Account

```
string[,] employeesArray = new string[5, 4]
{
    {"101","John","SE","10000"},
    {"102","Smith","SE","20000"},
    {"103","Dev","SSE","30000"},
    {"104","Pam","SE","40000"},
    {"105","Sara","SSE","50000"}
};
```

```
public void ProcessSalary(List<Employee> listEmployee)
{
        //Caclulate the Salary and
        //Deposit the Salary into Bank Account
}
```

Now the HR System will send the employee information in the form of a String Array to the Adapter. Then what this Adapter will do is, it will read the employee information from the string array and populate the employee object and then put each employee object into the List<Employee> collection, and then the Adapter will send the List<Employee> to the ProcessSalary method of Third Party Billing System. Then the ProcessSalary method calculates the Salary of each employee and deposits the salary into the Employee's bank account.

So, in this way, we can make two incompatible interfaces work together with the help of the Adapter Design Pattern in C#. Again the Adapter Design Pattern in C# can be implemented in two ways. They are as follows.

1. **Object Adapter Pattern**
2. **Class Adapter Pattern**

Step 1: Creating Employee Class

Employee.cs This class is going to be used by ThirdPartyBillingSystem (i.e. Adaptee) as well as by the Adapter.

```
namespace AdapterDesignPattern
{
   public class Employee
   {
      public int ID { get; set; }
      public string Name { get; set; }
      public string Designation { get; set; }
      public decimal Salary { get; set; }

      public Employee(int id, string name, string designation, decimal salary)
      {
         ID = id;
         Name = name;
         Designation = designation;
         Salary = salary;
      }
   }
}
```

Step2: Creating Adaptee

This is going to be a class that contains the functionality that is required by the client. However, this interface is not compatible with the client. So, create a class file with the name **ThirdPartyBillingSystem.cs**

This class is having the ProcessSalary method which takes a list of employees as an input parameter and then processes the salary of each employee.

```
using System;
using System.Collections.Generic;
namespace AdapterDesignPattern
{
    // The Adaptee contains some functionality that is required by the client.
    // But this interface is not compatible with the client code.
    public class ThirdPartyBillingSystem
    {
        //ThirdPartyBillingSystem accepts employee's information as a List to process each employee's salary
        public void ProcessSalary(List<Employee> listEmployee)
        {
            foreach (Employee employee in listEmployee)
            {
                Console.WriteLine("Rs." + employee.Salary + " Salary Credited to " + employee.Name + " Account");
            }
        }
    }
}
```

Step3: Creating ITarget interface

This is going to be the domain-specific interface that is going to be used by the client. So, create an interface with the name **ITarget.cs**
This class defines the abstract ProcessCompanySalary method which is going to be implemented by the Adapter. Again the client is going to use this method to process the salary.

```
namespace AdapterDesignPattern
{
    // The ITarget defines the domain-specific interface used by the client code.
    // This interface needs to be implemented by the Adapter.
    // The client can only see this interface i.e. the class which implements the ITarget interface.
    public interface ITarget
    {
        void ProcessCompanySalary(string[,] employeesArray);
    }
}
```

Step4: Create an Adapter

This class implements the ITarget interface and provides the implementation for the **ProcessCompanySalary** method. This class also has a reference to the ThirdPartyBillingSystem (Adaptee) object. The ProcessCompanySalary method receives the employee information as a string array and then converts the string array to a list of Employees and then calls the ProcessSalary method on the ThirdPartyBillingSystem (Adaptee) object by providing the list of employees as an argument.

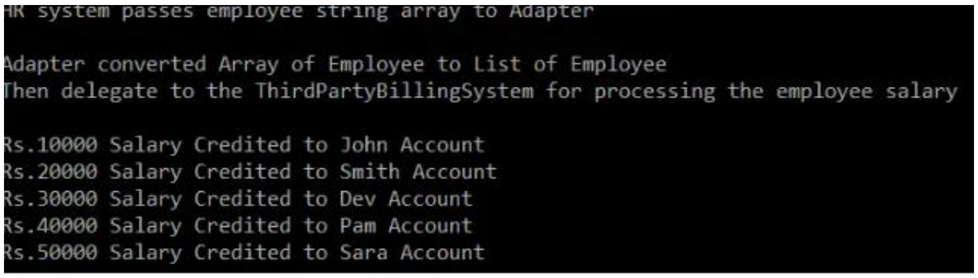
```
using System;
using System.Collections.Generic;
namespace AdapterDesignPattern
{
   // This is the class that makes two incompatible interfaces or systems work together.
   // The Adapter makes the Adaptee's interface compatible with the Target's interface.
   public class EmployeeAdapter : ITarget
   {
      //To use Object Adapter Design Pattern, we need to create an object of ThirdPartyBillingSystem
      ThirdPartyBillingSystem thirdPartyBillingSystem = new ThirdPartyBillingSystem();

      //The following will accept the employees in the form of string array
      //Then convert the employee string array to List of Employees
      //After conversation, it will call the Adaptee's Method to Process the Salaries
      public void ProcessCompanySalary(string[,] employeesArray)
      {
         string Id = null;
         string Name = null;
         string Designation = null;
         string Salary = null;

         List<Employee> listEmployee = new List<Employee>();

         for (int i = 0; i < employeesArray.GetLength(0); i++)
         {
            for (int j = 0; j < employeesArray.GetLength(1); j++)
            {
               if (j == 0)
               {
                  Id = employeesArray[i, j];
               }
               else if (j == 1)
               {
                  Name = employeesArray[i, j];
               }
               else if (j == 2)
               {
                  Designation = employeesArray[i, j];
               }
               else
               {
                  Salary = employeesArray[i, j];
               }
            }

            listEmployee.Add(new Employee(Convert.ToInt32(Id), Name, Designation, Convert.ToDecimal(Salary)));
         }

         Console.WriteLine("Adapter converted Array of Employee to List of Employee");
         Console.WriteLine("Then delegate to the ThirdPartyBillingSystem for processing the employee salary\n");
         thirdPartyBillingSystem.ProcessSalary(listEmployee);
      }
   }
}
```

Step5: Client

Then we create an instance of **EmployeeAdapter** and call the **ProcessCompanySalary** method by passing the string array as an argument. With the help of the Adapter (i.e. EmployeeAdapter object), now the Client and the Third Party Billing System work together.

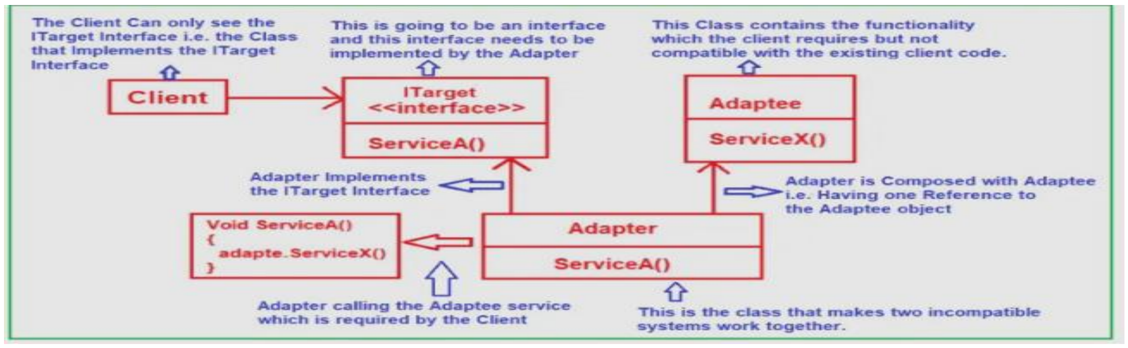
```
using System;
namespace AdapterDesignPattern
{
   //Client
   //The Client is Incompatible with ThirdPartyBillingSystem
   class Program
   {
      static void Main(string[] args)
      {
         //Storing the Employees Data in a String Array
         string[,] employeesArray = new string[5, 4]
         {
            {"101","John","SE","10000"},
            {"102","Smith","SE","20000"},
            {"103","Dev","SSE","30000"},
            {"104","Pam","SE","40000"},
            {"105","Sara","SSE","50000"}
         };

         //The EmployeeAdapter Makes it possible to work with Two Incompatible Interfaces
         Console.WriteLine("HR system passes employee string array to Adapter\n");
         ITarget target = new EmployeeAdapter();
         target.ProcessCompanySalary(employeesArray);

         Console.Read();
      }
   }
}
```

Output:



```
HR system passes employee string array to Adapter

Adapter converted Array of Employee to List of Employee
Then delegate to the ThirdPartyBillingSystem for processing the employee salary

Rs.10000 Salary Credited to John Account
Rs.20000 Salary Credited to Smith Account
Rs.30000 Salary Credited to Dev Account
Rs.40000 Salary Credited to Pam Account
Rs.50000 Salary Credited to Sara Account
```

UML Diagram of Object Adapter Design Pattern in C#:


The Client Can only see the ITarget Interface i.e. the Class that Implements the ITarget Interface
This is going to be an interface and this interface needs to be implemented by the Adapter
This Class contains the functionality which the client requires but not compatible with the existing client code.
Client
ITarget <<interface>>
ServiceA()
Adaptee
ServiceX()
Adapter Implements the ITarget Interface
Adapter is Composed with Adaptee i.e. Having one Reference to the Adaptee object
Void ServiceA() { adapte.ServiceX() }
Adapter
ServiceA()
Adapter calling the Adaptee service which is required by the Client
This is the class that makes two incompatible systems work together.

Strategy Design Pattern

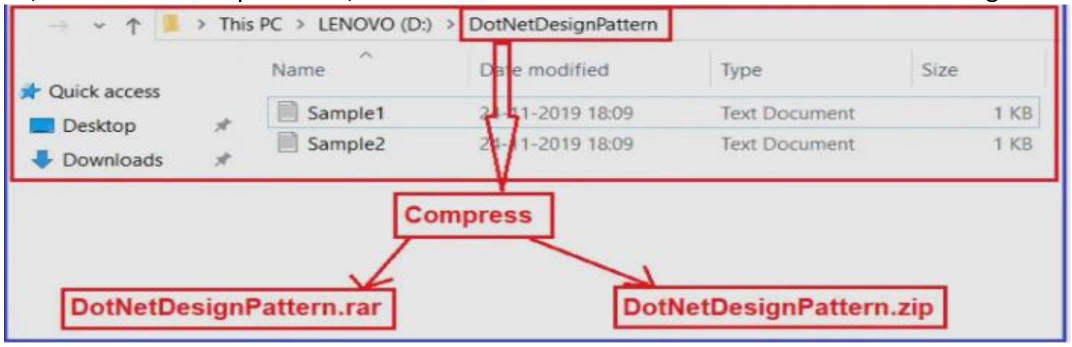
What is the Strategy Design Pattern?

The Strategy Design Pattern is used when we have multiple algorithms (solutions) for a specific task and the client decides on the actual implementation to be used at runtime. In simple words, we can say that the Strategy Design Pattern (also called Policy Pattern) attempts to solve the issue where you need to provide multiple solutions for the same problem so that one can be selected at runtime.

Understanding the Strategy Design Pattern with Real-time Example:

Let us understand the Strategy Design Pattern in C# using one real-time example. Please have a look at the following image. As you can see, in my D drive I have a folder called DotNetDesignPattern and within that folder, multiple text files are there. My business requirement is, I have to compress this DotNetDesignPattern folder and send the compressed file to the client.

For this requirement, I have two solutions. The first solution is to compress the folder into RAR format and send it to the client and the second solution is to compress the folder into ZIP format and sends it to the client. So, for the above requirement, I have two solutions which are shown in the below image.



How to Implement the Strategy Design Pattern in C#?

Step 1: Creating Strategy Interface

Create an interface with the name **ICompression.cs**

The Strategy Interface declares the methods that are common to all supported versions of the algorithm. The Context object is going to use this Strategy Interface to call the algorithm defined by the Concrete Strategies

```
namespace StrategyDesignPattern
{
   // Strategy Interface
   // The Strategy Interface declare methods that are common to all supported versions of the algorithm.
   // The Context Object is going to use this Strategy Interface to call the algorithm defined by Concrete Strategies.
   public interface ICompression
   {
      void CompressFolder(string compressedArchiveFileName);
   }
}
```

Step 2: Creating Concrete Strategies

The Concrete Strategies are the classes that implement the Strategy interface. Each Concrete Strategy by which we will compress a file item must be implementing the method CompressFolder of the ICompression interface. Let's create two Concrete Strategy classes as per our business requirement.

RarCompression:

Create a class file with the name **RarCompression.cs** and then copy and paste the following code into it. The following RarCompression Concrete Strategy implements the Strategy Interface and Implement the CompressFolder Method. This algorithm returns the Folder in RAR format.

```
using System;
namespace StrategyDesignPattern
{
   // Concrete Strategy A
   // The following RarCompression Concrete Strategy implement the Strategy Interface and
   // Implement the CompressFolder Method.
   public class RarCompression : ICompression
   {
      public void CompressFolder(string compressedArchiveFileName)
      {
         Console.WriteLine("Folder is compressed using Rar approach: '" + compressedArchiveFileName
             + ".rar' file is created");
      }
   }
}
```

ZipCompression:

Create a class file with the name **ZipCompression.cs** . The following ZipCompression Concrete Strategy implements the Strategy Interface and Implement the CompressFolder Method. This algorithm returns the Folder in ZIP format.

```
using System;
namespace StrategyDesignPattern
{
   // Concrete Strategy B
   // The following ZipCompression Concrete Strategy implement the Strategy Interface and
   // Implement the CompressFolder Method.
   public class ZipCompression : ICompression
   {
      public void CompressFolder(string compressedArchiveFileName)
      {
         Console.WriteLine("Folder is compressed using zip approach: '" + compressedArchiveFileName
             + ".zip' file is created");
      }
   }
}
```

Step 3: Creating Context

Create a class file with the name **CompressionContext.cs** . This context class contains a property that holds the reference of a Concrete Strategy object. This property will be set at run-time by the client according to the algorithm that is required.

```
namespace StrategyDesignPattern
{
   // The Context Provides the interface which is going to be used by the Client.
   public class CompressionContext
   {
      // The Context has a reference to one of the Strategy objects.
      // The Context does not know the concrete class of a strategy.
```

```
        // It should work with all strategies via the Strategy interface.
        private ICompression Compression;

        //Initializing the Strategy Object i.e. Compression using Constructor
        public CompressionContext(ICompression Compression)
        {
            // The Context accepts a strategy through the constructor,
            // but also provides a setter method to change the strategy at runtime
            this.Compression = Compression;
        }

        //The Context allows replacing a Strategy object at runtime.
        public void SetStrategy(ICompression Compression)
        {
            this.Compression = Compression;
        }

        // The Context delegates the work to the Strategy object instead of
        // implementing multiple versions of the algorithm on its own.
        public void CreateArchive(string compressedArchiveFileName)
        {
            //The CompressFolder method is going to be invoked based on the strategy object
            Compression.CompressFolder(compressedArchiveFileName);
        }
    }
}
```

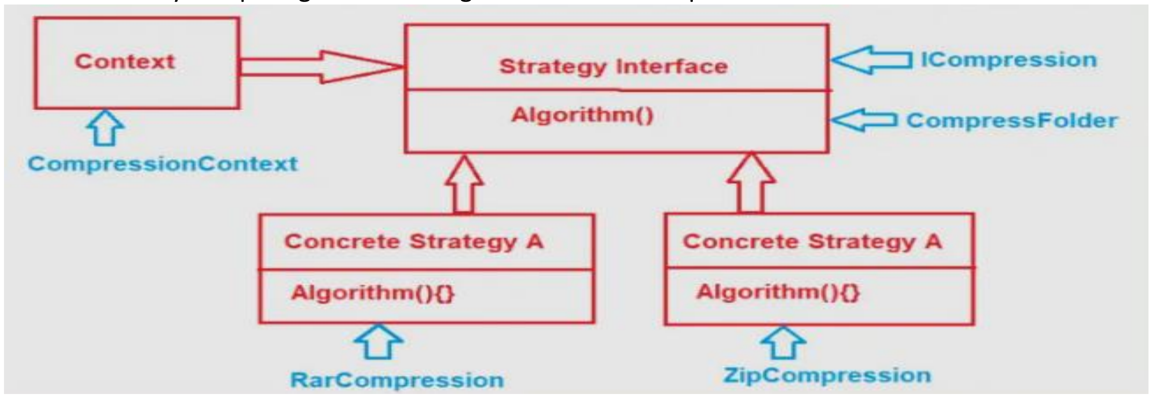
Step 4: Client

The Main method of the Program class is going to be the client. So, modify the Main method of the Program class as shown below. The following Client Code is self-explained, so please go through the comment lines for a better understanding.

```
using System;
namespace StrategyDesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            // The client code picks a concrete strategy and passes it to the context.
            // The client should be aware of the differences between strategies in order to make the right choice.

            //Create an instance of ZipCompression Strategy
            ICompression strategy = new ZipCompression();

            //Pass ZipCompression Strategy as an argument to the CompressionContext constructor
            CompressionContext ctx = new CompressionContext(strategy);
            ctx.CreateArchive("DotNetDesignPattern");

            //Changing the Strategy using SetStrategy Method
            ctx.SetStrategy(new RarCompression());
            ctx.CreateArchive("DotNetDesignPattern");

            Console.Read();
        }
    }
}
```

Strategy Design Pattern UML or Class Diagram:

the Class Diagram or UML Diagram of the Strategy Design Pattern and understand the different components involved in it by comparing the UML Diagram with our Example



1. **Strategy:** The Strategy declares an interface that is going to be implemented by all supported algorithms. Here, algorithms mean methods, and these methods are going to be implemented by the concrete strategy classes. In our example, it is the CompressFolder method of the ICompression interface.
2. **ConcreteStrategy:** These are going to be concrete classes and they must implement the Strategy (ICompression) interface and provide implementations for the algorithm i.e. implementing the interface methods. In our example, it is the RarCompression and ZipCompression classes.
3. **Context:** This is going to be a class that maintains a reference to a Strategy object, and then uses that reference to call the algorithm defined by a particular ConcreteStrategy (i.e. either RarCompression or ZipCompression). In our example, it is the CompressionContext class.

When do we need to use the Strategy Design Pattern in Real-Time Applications?

1. When there are multiple solutions for a given task and the selection criteria of a solution are defined at run-time.
2. When you want different variants of an algorithm.
3. When many related classes differ only in their behavior.
4. When a class defines many behaviors and these appear as multiple conditional statements in its operations. Instead of many conditional statements, move-related conditional branches into their own strategy class.
5. When an algorithm uses data that the client shouldn't know about. Use the Strategy Design Pattern to avoid exposing complex and algorithm-specific data structures.

What is the null object pattern?

The null object pattern is a behavioral software design pattern that is used to encapsulate the handling of null values in an object-oriented programming language. It is used to help improve code readability and maintainability by eliminating the need for explicit null checks.

The null object pattern defines an abstract class that represents a null value. A concrete subclass is then created that inherits from the abstract class and provides concrete implementations for all of the methods. This concrete subclass is then used throughout the code base whenever a null value needs to be represented.

To implement the null object pattern in C#, we'll follow the three steps outlined below.

1. Define an interface or an abstract class.
2. Provide concrete implementations of the methods of the abstract class or the interface in a class that extends either of them.
3. Define a null implementation by overriding the methods of the abstract base class or the interface and returning appropriate values for your null object.

```
public abstract class AbstractProduct
  {
    public abstract int Id { get; set; }
    public abstract string Name { get; set; }
    public abstract string GetProductDetails();
  }
```

Implement methods of the abstract class in C#

Next, create class named Product

```
namespace NullObjectPattern
{
  public class NullProduct : AbstractProduct
  {
    public override int Id
    {
      get;  set;
    }
    public override string Name
    {
      get; set;
    }
    public override string GetProductDetails()
    {
      return $"Product Name: {Name}";
    }
  }
}
```

create another new class named Product in a file named Product.cs

```
namespace NullObjectPattern
{
   public class Product : AbstractProduct
   {
      public override int Id
      {  get;  set;
      }
      public override string Name
      {  get; set;
      }
      public override string GetProductDetails()
      {
         return $"Product Id: {Id}, Product Name: {Name}";
      }
   }
}
```

Now, create the ProductRepository class used to work with the product data of our example application. ProductRepository contains a method named GetProduct that accepts the product ID as a parameter and returns either a product instance (if the product is found) or an instance of NullProduct if no product is found.

```
namespace NullObjectPattern
{
   public class ProductRepository
   {
      List<Product> products = new List<Product>();
      NullProduct? NotFound = new() { Id = -1, Name = "Not Available" };
      public ProductRepository()
      {
         products.Add(
            new Product()
            {
               Id = 1,
               Name = "DELL Laptop"
            });
         products.Add(
            new Product()
            {
               Id = 2,
               Name = "Lenovo Laptop"
            });
      }
      public AbstractProduct GetProduct(int id)
      {
         AbstractProduct product = products.Find(x => x.Id == id);
         return product ?? NotFound;
      }
   }
}
```

Execute the application

```
namespace NullObjectPattern
{
   class Program
   {
      static void Main(string[] args)
      {
         //// -------------When Product is not null----------
         ProductRepository productRepository = new ProductRepository();
         var product = productRepository.GetProduct(1);
         Console.WriteLine(product.GetProductDetails());
         // -------------When Product  null----------
         ProductRepository productRepository = new ProductRepository();
         var product = productRepository.GetProduct(3);
         Console.WriteLine(product.GetProductDetails());
      }
   }
}
```

The Template Method design pattern

The Template Method pattern is best used when you have an algorithm consisting of certain steps and you want to allow for different implementations of these steps. The implementation details of each step can vary but the structure and order of the steps are enforced.

A good example is games:

1. Set up the game
2. Take turns
3. Game is over
4. Display the winner

A large number of games can implement this algorithm, such as Monopoly, Chess, card games etc. Each game is set up and played in a different way but they follow the same order.

Note that a prerequisite for this pattern to be applied properly is the rigidness of the algorithm steps. The steps must be known and well defined. The pattern relies on inheritance, rather than composition, and merging two child algorithms into one can prove difficult. If you find that the Template pattern is too limiting in your application then consider the Strategy or the Decorator patterns.

This pattern helps to implement the so-called Hollywood principle: Don’t call us, we’ll call you. It means that high level components, i.e. the superclasses, should not depend on low-level ones, i.e. the implementing subclasses. A base class with a template method is a high level component and clients should depend on this class. The base class will include one or more template method that the subclasses implement,

. Add a class called OrderShipment:

```
public abstract class OrderShipment
    {
        public string ShippingAddress { get; set; }
        public string Label { get; set; }
        public void Ship(TextWriter writer)
        {
            VerifyShippingData();
            GetShippingLabelFromCarrier();
            PrintLabel(writer);
        }

        public virtual void VerifyShippingData()
        {
            if (String.IsNullOrEmpty(ShippingAddress))
            {
                throw new ApplicationException("Invalid address.");
            }
        }
        public abstract void GetShippingLabelFromCarrier();
        public virtual void PrintLabel(TextWriter writer)
        {
            writer.Write(Label);
        }
    }
```

The template method that implements the order of the steps is Ship. It calls three methods in a specific order. Two of them – VerifyShippingData and PrintLabel are virtual and have a default implementation. They can of course be overridden. The third method, i.e. GetShippingLabelFromCarrier is the abstract method that the base class cannot implement. The superclass has no way of knowing what a service-specific shipping label looks like – it is delegated to the implementations. We'll simulate two services, UPS and FedEx:

```
class FedExOrderShipment : OrderShipment
    {
        public override void GetShippingLabelFromCarrier()
        {
            // Call FedEx Web Service
            Label = String.Format("FedEx:[{0}]", ShippingAddress);
        }
    }

class UpsOrderShipment : OrderShipment
    {
        public override void GetShippingLabelFromCarrier()
        {
            // Call UPS Web Service
            Label = String.Format("UPS:[{0}]", ShippingAddress);
        }
    }
```

The implementations should be quite straighforward: they create service-specific shipping labels and set those values to the Label property. There's of course nothing stopping the concrete classes from overriding any other step in the algorithm. Adding new shipping services is very easy: just create a new implementation. Let's see how a client would communicate with the services:

The Chain of Responsibility pattern

The Chain of Responsibility is an ordered chain of message handlers that can process a specific type of message or pass the message to the next handler in the chain. This pattern revolves around messaging between a sender and one more receivers.

The example also showcases the traits of the pattern:

- The Sender is only aware of the first receiver
- Each receiver only knows of the next receiver down the messaging chain
- Receivers can process the Message or send it down the chain
- The Sender will have no knowledge about which Receiver received the message
- The first receiver that was able to process the message terminates the chain
- The order of the receiver list matters

Demo

In the demo we'll simulate the hierarchy of a company: an employee would like to make a large expenditure so he asks his manager. The manager is not entitled to approve the large sum and sends the request forward to the VP. The VP is not entitled either to approve the request so sends it to the President. The President is the highest authority in the hierarchy who will either approve or disapprove the request and sends the response back to the original employee.

We'll start with the abstraction for an expense report, IExpenseReport:

```
public interface IExpenseReport
    { Decimal Total { get;}
    }
```

The IExpenseApprover interface represents any object that is entitled to approve expense reports:

```
    public interface IExpenseApprover
        {
            ApprovalResponse ApproveExpense(IExpenseReport expenseReport);
        }
```

…where ApprovalResponse is an enumeration:

```
    public enum ApprovalResponse
        {
            Denied,
            Approved,
            BeyondApprovalLimit,
        }
```

The concrete implementation of the IExpenseReport is very straightforward:

```
     public class ExpenseReport : IExpenseReport
         {
             public ExpenseReport(Decimal total)
             {
                 Total = total;
             }

             public decimal Total
             {
                 get;
                 private set;
             }
         }
```

The Employee class implements the IExpenseApprover interface:

```
     public class Employee : IExpenseApprover
         {
             public Employee(string name, Decimal approvalLimit)
             {
                Name = name;
                _approvalLimit = approvalLimit;
            }

            public string Name { get; private set; }

            public ApprovalResponse ApproveExpense(IExpenseReport
    expenseReport)
            {
                return expenseReport.Total > _approvalLimit
                        ? ApprovalResponse.BeyondApprovalLimit
                        : ApprovalResponse.Approved;
            }

            private readonly Decimal _approvalLimit;
        }



static void Main(string[] args)
        {
            List<Employee> managers = new List<Employee>
                                          {
                                              new Employee("William Worker",
Decimal.Zero),
                                              new Employee("Mary Manager", new
Decimal(1000)),
                                              new Employee("Victor Vicepres",
new Decimal(5000)),
                                              new Employee("Paula President",
new Decimal(20000)),
                                          };

            Decimal expenseReportAmount;
            while (ConsoleInput.TryReadDecimal("Expense report amount:", out
expenseReportAmount))
            {
                IExpenseReport expense = new
ExpenseReport(expenseReportAmount);

                bool expenseProcessed = false;

                foreach (Employee approver in managers)
                {
                    ApprovalResponse response =
approver.ApproveExpense(expense);

                    if (response != ApprovalResponse.BeyondApprovalLimit)
                    {
                        Console.WriteLine("The request was {0}.", response);
                        expenseProcessed = true;
                        break;
                    }
                }

                if (!expenseProcessed)
                {
                    Console.WriteLine("No one was able to approve your
expense.");
                }
            }
        }
```
where ConsoleInput is a helper class that looks as follows:
```
    public static class ConsoleInput
        {
            public static bool TryReadDecimal(string prompt, out Decimal value)
            {
                value = default(Decimal);

                while (true)
                {
                    Console.WriteLine(prompt);
                    string input = Console.ReadLine();

                    if (string.IsNullOrEmpty(input))
                    {
                        return false;
                    }

                    try
                    {
                        value = Convert.ToDecimal(input);
                        return true;
                    }
                    catch (FormatException)
                    {
                        Console.WriteLine("The input is not a valid
    decimal.");
                    }
                    catch (OverflowException)
                    {
                        Console.WriteLine("The input is not a valid
    decimal.");
                    }
                }
            }
        }
```

As you see the constructor needs a name and an approval limit. The implemented ApproveExpense method simply checks if the total value of the expense report is above or below the approval limit. If total is lower than the limit, then the expense is approved, otherwise the method indicates that the total is too much for the employee to approve.

Build and run the application. Enter 5000 in the console and you'll see that the expense was approved. You'll recall that the VP had an approval limit of 5000 so it was that employee in the chain to approve. Enter 50000 and you'll see that nobody was able to approve the expense because it exceeds the limit of every one of them

What is wrong with this implementation?

Imagine that you as an employee should not ask each one of the managers above you for a yes or no answer. You should only have to turn to your boss who in turn will ask his or her boss etc. Our code should reflect this.

In order to achieve that we need to insert a new interface:

The problem is that the caller is responsible for iterating through the list. This means that the logic of handling expense reports is encapsulated at the wrong level. Imagine that you as an employee should not ask each one of the managers above you for a yes or no answer. You should only have to turn to your boss who in turn will ask his or her boss etc. Our code should reflect this.
In order to achieve that we need to insert a new interface:

```
    public interface IExpenseHandler
        {
            ApprovalResponse Approve(IExpenseReport expenseReport);
            void RegisterNext(IExpenseHandler next);
        }
```

The RegisterNext method registers the next approver in the chain. It means that if I cannot approve the expense then I should go and ask the next person in line.
This interface represents a single link in the chain of responsibility.
The IExpenseHandler interface is implemented by the ExpenseHandler class:

```
public class ExpenseHandler : IExpenseHandler
    {
        private readonly IExpenseApprover _approver;
        private IExpenseHandler _next;

        public ExpenseHandler(IExpenseApprover expenseApprover)
        {
            _approver = expenseApprover;
            _next = EndOfChainExpenseHandler.Instance;
        }

        public ApprovalResponse Approve(IExpenseReport expenseReport)
        {
            ApprovalResponse response = _approver.ApproveExpense(expenseReport);

            if (response == ApprovalResponse.BeyondApprovalLimit)
            {
                return _next.Approve(expenseReport);
            }

            return response;
        }

        public void RegisterNext(IExpenseHandler next)
        {
            _next = next;
        }
    }
```

The constructor makes sure that there is always a special end of chain Employee in the approval chain through the EndOfChainExpenseHandler class. The Approve method receives an expense report. We ask the approver if they are able to approver the expense. If not, then we go to the next person in the hierarchy, i.e. to the “next” variable.
The implementation of the EndOfChainExpenseHandler class follows below. It also implements the IExpenseHandler method and it represents – as the name implies – the last member in the approval hierarchy. Its Instance property returns this special member of the chain according to the singleton pattern – more on that here.

```
    public class EndOfChainExpenseHandler : IExpenseHandler
        {
            private EndOfChainExpenseHandler() { }

            public static EndOfChainExpenseHandler Instance
            {
                get { return _instance; }
            }

            public ApprovalResponse Approve(IExpenseReport expenseReport)
            {
                return ApprovalResponse.Denied;
            }

            public void RegisterNext(IExpenseHandler next)
            {
                throw new InvalidOperationException("End of chain handler must be the end
    of the chain!");
            }

            private static readonly EndOfChainExpenseHandler _instance = new
    EndOfChainExpenseHandler();
        }
```

The purpose of this class is to make sure that if the last person in the hierarchy, i.e. the President, is unable to approve the report then it is not passed on to a null reference – as there's nobody above the President – but that there's an automatic message handler that gives some default answer

```
static void Main(string[] args)
{
    ExpenseHandler william = new ExpenseHandler(new Employee("William Worker",
Decimal.Zero));
    ExpenseHandler mary = new ExpenseHandler(new Employee("Mary Manager", new
Decimal(1000)));
    ExpenseHandler victor = new ExpenseHandler(new Employee("Victor Vicepres",
new Decimal(5000)));
    ExpenseHandler paula = new ExpenseHandler(new Employee("Paula President",
new Decimal(20000)));

    william.RegisterNext(mary);
    mary.RegisterNext(victor);
    victor.RegisterNext(paula);

    Decimal expenseReportAmount;
    if (ConsoleInput.TryReadDecimal("Expense report amount:", out
expenseReportAmount))
    {
        IExpenseReport expense = new ExpenseReport(expenseReportAmount);
        ApprovalResponse response = william.Approve(expense);
        Console.WriteLine("The request was {0}.", response);
    }
        Console.ReadKey();
}
```

The Composite pattern

The Composite pattern deals with putting individual objects together to form a whole. In mathematics the relationship between the objects and the composite object they build can be described by a part-whole hierarchy. The ingredient objects are the parts and the composite is the whole.

A real life example is sending emails. If you want to send an email to all developers in your organisation one option is that you type in the names of each developer in the ‘to’ field. This is of course not efficient. Fortunately we can construct recipient groups, such as Developers. If you then also want to send the email to another person outside the Developers group you can simply put their name in the ‘to’ box along with Developers. We treat both the group and the individual emails in a uniform way. We can insert both groups and individual emails in the ‘to’ box. We rely on the email engine to take the group apart and send the email to each recipient in that group. We don’t really care how it’s done – apart from a couple network geeks I guess.

Demo
We will first build a demo application that does not use the pattern and then we’ll refactor it. We’ll simulate a game where play money is split among the players in a group if they manage to kill a monster.