***Design Patterns***

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

***Uses of Design Patterns***

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

***Creational design patterns***

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

1. **Abstract Factory**:  Creates an instance of several families of classes. Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
2. **Builder**: Separates object construction from its representation. Separate the construction of a complex object from its representation so that the same construction processes can create different representations.
3. **Factory Method**: Creates an instance of several derived classes. Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
4. **Prototype**: A fully initialized instance to be copied or cloned. Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
5. **Singleton:** A class of which only a single instance can exist. Ensure a class only has one instance, and provide a global point of access to it.

***Structural design patterns***

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

1. **Adapter**: Match interfaces of different classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
2. **Bridge**: Separates an object’s interface from its implementation. Decouple an abstraction from its implementation so that the two can vary independently.
3. **Composite:**A tree structure of simple and composite objects. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
4. **Decorator**: Add responsibilities to objects dynamically.  Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
5. **Facade:**A single class that represents an entire subsystem. Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use.
6. **Flyweight:**A fine-grained instance used for efficient sharing. Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context — it’s indistinguishable from an instance of the object that’s not shared.
7. **Proxy:**An object representing another object. Provide a surrogate or placeholder for another object to control access to it.

***Behavioral design patterns***

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

1. **Chain of Responsibility**: A way of passing a request between a chain of objects. Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
2. **Command:**Encapsulate a command request as an object. Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
3. **Interpreter**: A way to include language elements in a program. Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
4. **Iterator**: Sequentially access the elements of a collection. Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
5. **Mediator**: Defines simplified communication between classes. Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
6. **Memento**: Capture and restore an object's internal state. Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.
7. **Observer**: A way of notifying change to a number of classes. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
8. **State**: Alter an object's behavior when its state changes. Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
9. **Strategy**: Encapsulates an algorithm inside a class. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
10. **Template:**Defer the exact steps of an algorithm to a subclass. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.
11. **Visitor:** Defines a new operation to a class without change. Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

***What is the difference between Repository Pattern, Factory Pattern and Singleton Pattern?***

1. The factory pattern deals with how an object is created. It gets classified under the creational pattern. Factory pattern defines an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. We can go for a factory pattern when we need for
   1. Object creation without exposing it to client
   2. Refer to newly created objects through interface.
2. Singleton pattern ensures only one instance of the type always be available for the client(s).Let us look into the below code
   1. The constructor is defined as private ensuring that no instance of the class can be created. Singleton keeps common data in only one place and this can be done by using the "static" keyword where the property "Singleton" is defined.
   2. Next checking if the property value is null or not. If yes then we are creating an instance and if no then we are returning back the old instance only. This is done inside the CreateInstance method. The client invocation part is as under var \_singletonInstance = Singleton.CreateInstance();
3. Repository pattern comes into use when we have (at a minimum) at least one of the following requirements

a) We want to centrally manage the data by applying logic and rules while dealing with data that comes from varied location.

b) We need to use business entities that are strongly typed. These helps to find errors at comile time rather at runtime.

c) Isolate data layer for unit testing.

d) Want to have a caching mechanism in place to hold the data in order to improve the application performance.

e) Separate business logic and data layer logic for better code maintainability.

***Why should a Singleton class be serialized only once ?***

If a Singleton class is serialized and then Deserialized multiple times, there will be multiple objects which will violate the principles of Singleton pattern. Therefore it should be serialized and Deserialized only once !

***Singleton is also called as AntiPattern, why ?***

It's very hard to create a subclass, or to create a mock object for a Singleton class. Singleton makes the unit testing harder. Singleton hide the dependencies and make the classes tightly coupled with each other.

***What is the importance of Design Pattern?***

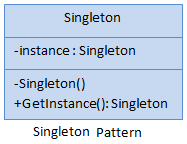
Here is a list of points which should be consider as some of the advantages of using design pattern

|  |  |
| --- | --- |
| * It brings loose coupling * It reduces object dependency * Components can be pluggable. * Platform independent | * It normalizes the communication between the client and the concrete classes. * Enhances Reusability Brings a fruitful framework Independent of language * Simplifies the application handshaking * Abstract the complex details from the client . * Can be applied in cross platform communication. |

***Singleton Design Pattern - C#***

Singleton pattern falls under Creational Design Pattern. It is pattern is one of the simplest design patterns. This pattern ensures that a class has only one instance and provides a global point of access to it.

***Singleton Pattern - UML Diagram & Implementation***



1. *//eager initialization of singleton*

public class Singleton

{

private static Singleton instance = new Singleton(); private Singleton() { }

public static Singleton GetInstance { get { return instance; } }

}

1. *////lazy initialization of singleton*

public class Singleton

{

private static Singleton instance = null; private Singleton() { }

public static Singleton GetInstance { get { if (instance == null) instance = new Singleton(); return instance; } }

}

1. *////Thread-safe (Double-checked Locking) initialization of singleton*

public class Singleton

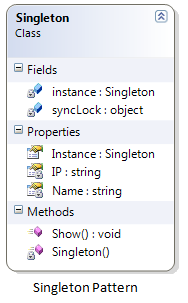
{

private static Singleton instance = null; private Singleton() { } private static object lockThis = new object();

public static Singleton GetInstance { get { lock (lockThis) { if (instance == null) instance = new Singleton(); return instance; } } }

}

Singleton Pattern - Example



*/// <summary> /// The 'Singleton' class*

public class Singleton

{ *// .NET guarantees thread safety for static initialization*

private static Singleton instance = null; private string Name{get;set;} private string IP{get;set;}

private Singleton() { *//To DO: Remove below line*  Console.WriteLine("Singleton Intance"); Name = "Server1"; IP = "192.168.1.23"; }

*// Lock synchronization object*

private static object syncLock = new object();

*// Support multithreaded applications through 'Double checked locking' pattern which (once the instance exists) avoids locking each*

*// time the method is invoked*

public static Singleton Instance { get {lock (syncLock) { if (Singleton.instance == null) Singleton.instance = new Singleton();

return Singleton.instance; } }

}

public void Show() { Console.WriteLine("Server Information is : Name={0} & IP={1}", IP, Name); }}

*/// <summary> /// Singleton Pattern Demo ///*

class Program { static void Main(string[] args) { Singleton.Instance.Show(); Singleton.Instance.Show();  Console.ReadKey(); }

***Singleton Pattern Demo - Output***

http://www.dotnettricks.com/img/designpatterns/Singleton2.png

***When to use it?***

1. Exactly one instance of a class is required.
2. Controlled access to a single object is necessary

***Factory Method Design Pattern***

Factory method pattern falls under Creational Pattern. It is used to create objects. People usually use this pattern as the standard way to create objects. In Factory pattern, we create object without exposing the creation logic. In this pattern, an interface is used for creating an object, but let subclass decide which class to instantiate. The creation of object is done when it is required. The Factory method allows a class later instantiation to subclasses.

|  |  |
| --- | --- |
| ***Factory Method Pattern - UML Diagram & Implementation***http://www.dotnettricks.com/img/designpatterns/factory.png | The classes, interfaces and objects in the above UML class diagram are as follows:   1. **Product:** This is an interface for creating the objects. 2. **ConcreteProduct:** This is a class which implements the Product interface. 3. **Creator:** This is an abstract class and declares the factory method, which returns an object of type Product. 4. **ConcreteCreator:** This is a class which implements the Creator class and overrides the factory method to return an instance of a ConcreteProduct. |

***C# - Implementation Code***

interface Product {}  class ConcreteProductA : Product{}  class ConcreteProductB : Product{}

abstract class Creator { public abstract Product FactoryMethod(string type);}

class ConcreteCreator : Creator{ public override Product FactoryMethod(string type) { switch (type) { case "A": return new ConcreteProductA(); case "B": return new ConcreteProductB(); }}}

***Factory Method Pattern - Example***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/factory1.png | The classes, interfaces and objects in the above class diagram can be identified as follows:   1. **IFactory** - Interface 2. **Scooter & Bike**- Concreate Product classes 3. **VehicleFactory**- Creator 4. **ConcreteVehicleFactory**- Concreate Creator |

namespace Factory

{

public interface IFactory { void Drive(int miles); }

public class Scooter : IFactory { public void Drive(int miles) { Console.WriteLine("Drive the Scooter : " + miles.ToString() + "km"); } }

  public class Bike : IFactory { public void Drive(int miles) { Console.WriteLine("Drive the Bike : " + miles.ToString() + "km"); } }

  public abstract class VehicleFactory { public abstract IFactory GetVehicle(string Vehicle);  }

public class ConcreteVehicleFactory : VehicleFactory {

public override IFactory GetVehicle(string Vehicle) { switch (Vehicle) { case "Scooter": return new Scooter(); case "Bike": return new Bike(); } }

}

static void Main(string[] args)

{

VehicleFactory factory = new ConcreteVehicleFactory();

IFactory scooter = factory.GetVehicle("Scooter"); scooter.Drive(10); IFactory bike = factory.GetVehicle("Bike"); bike.Drive(20); } }

}

***Factory Pattern Demo - Output***

http://www.dotnettricks.com/img/designpatterns/factory2.png

***When to use it?***

1. Subclasses figure out what objects should be created.
2. Parent class allows later instantiation to subclasses means the creation of object is done when it is required.
3. The process of objects creation is required to centralize within the application.
4. A class (creator) will not know what classes it will be required to create.

***Abstract Factory Design Pattern***

Abstract Factory method pattern falls under Creational Pattern. Abstract Factory patterns acts a super-factory which creates other factories. This pattern is also called as Factory of factories. In Abstract Factory pattern an interface is responsible for creating a set of related objects, or dependent objects without specifying their concrete classes. Internally, Abstract Factory use Factory design pattern for creating objects. It may also use Builder design pattern and prototype design pattern for creating objects. It completely depends upon your implementation for creating objects.

***Abstract Factory Pattern - UML Diagram & Implementation***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/abstractfactory.png | The classes, interfaces and objects in the above UML class diagram are as follows:   1. **AbstractFactory** This is an interface which is used to create abstract product 2. **ConcreteFactory** This is a class which implements the AbstractFactory interface to create concrete products. 3. **AbstractProduct** This is an interface which declares a type of product. 4. **ConcreteProduct** This is a class which implements the AbstractProduct interface to create product. 5. **Client** This is a class which use AbstractFactory and AbstractProduct interfaces to create a family of related objects. |

public interface AbstractFactory { AbstractProductA CreateProductA(); AbstractProductB CreateProductB();}

public class ConcreteFactoryA : AbstractFactory

{

public AbstractProductA CreateProductA() { return new ProductA1(); } public AbstractProductB CreateProductB() { return new ProductB1(); }

}

public class ConcreteFactoryB : AbstractFactory

{

public AbstractProductA CreateProductA() { return new ProductA2(); } public AbstractProductB CreateProductB() { return new ProductB2(); }

}

public interface AbstractProductA { } public class ProductA1 : AbstractProductA { } public class ProductA2 : AbstractProductA { }

public interface AbstractProductB { } public class ProductB1 : AbstractProductB { } public class ProductB2 : AbstractProductB { }

public class Client

{

private AbstractProductA \_productA; private AbstractProductB \_productB;

public Client(AbstractFactory factory) { \_productA = factory.CreateProductA(); \_productB = factory.CreateProductB(); }

}

***Abstract Factory Pattern - Example***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/abstractfactory1.png | The classes, interfaces and objects in the above class diagram can be identified as follows:   * **VehicleFactory** -AbstractFactory interface * **HondaFactory & HeroFactory**- Concrete Factories * **Bike & Scooter**- AbstractProduct interface * **Regular Bike, Sports Bike, Regular Scooter & Scooty**- Concreate Products * **VehicleClient**- Client |

interface VehicleFactory { Bike GetBike(string Bike); Scooter GetScooter(string Scooter);}

class HondaFactory : VehicleFactory */// The 'ConcreteFactory1' class.*

{

public Bike GetBike(string Bike) {switch (Bike){ case "Sports": return new SportsBike(); case "Regular": return new RegularBike(); } }

  public Scooter GetScooter(string Scooter) { switch (Scooter){ case "Sports": return new Scooty(); case "Regular": return new RegularScooter(); } }

}

class HeroFactory : VehicleFactory */// The 'ConcreteFactory2' class*

{

public Bike GetBike(string Bike) { switch (Bike) { case "Sports": return new SportsBike(); case "Regular": return new RegularBike(); } }

public Scooter GetScooter(string Scooter) { switch (Scooter) { case "Sports": return new Scooty(); case "Regular": return new RegularScooter(); } }

}

interface Bike{ string Name();} */// The 'AbstractProductA' interface*

interface Scooter{ string Name();} */// The 'AbstractProductB' interface*

class RegularBike : Bike{ public string Name() { return "Regular Bike- Name"; }} */// The 'ProductA1' class*

class SportsBike : Bike { public string Name() { return "Sports Bike- Name"; }} */// The 'ProductA2' class*

class RegularScooter : Scooter{ public string Name() { return "Regular Scooter- Name"; }} */// The 'ProductB1' class*

class Scooty : Scooter{ public string Name() { return "Scooty- Name"; }} */// The 'ProductB2' class*

class VehicleClient

{

Bike bike; Scooter scooter;

public VehicleClient(VehicleFactory factory, string type) { bike = factory.GetBike(type); scooter = factory.GetScooter(type);}

public string GetBikeName() { return bike.Name(); }

public string GetScooterName() { return scooter.Name(); }

 }

static void Main(string[] args)

{

VehicleFactory honda = new HondaFactory(); VehicleClient hondaclient = new VehicleClient(honda, "Regular");

Console.WriteLine("\*\*\*\* Honda \*\*\*\*\*\*\*"); Console.WriteLine(hondaclient.GetBikeName());Console.WriteLine(hondaclient.GetScooterName());

hondaclient = new VehicleClient(honda, "Sports"); Console.WriteLine(hondaclient.GetBikeName()); Console.WriteLine(hondaclient.GetScooterName());

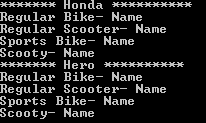
VehicleFactory hero = new HeroFactory(); VehicleClient heroclient = new VehicleClient(hero, "Regular");

Console.WriteLine("\*\*\*\* Hero \*\*\*\*"); Console.WriteLine(heroclient.GetBikeName()); Console.WriteLine(heroclient.GetScooterName());

heroclient = new VehicleClient(hero, "Sports"); Console.WriteLine(heroclient.GetBikeName()); Console.WriteLine(heroclient.GetScooterName());

}}

***Abstract Factory Pattern Demo - Output***



***When to use it?***

1. Create a set of related objects, or dependent objects which must be used together.
2. System should be configured to work with multiple families of products.
3. The creation of objects should be independent from the utilizing system.
4. Concrete classes should be decoupled from clients.

***Note***

1. Internally, Abstract Factory use Factory design pattern for creating objects. But it can also use Builder design pattern and prototype design pattern for creating objects. It completely depends upon your implementation for creating objects.
2. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
3. When Abstract Factory, Builder, and Prototype define a factory for creating the objects, we should consider the following points :
   1. Abstract Factory use the factory for creating objects of several classes.
   2. Builder use the factory for creating a complex object by using simple objects and a step by step approach.
   3. Prototype use the factory for building a object by copying an existing object.

***Builder Design Pattern***

Builder pattern falls under Creational Pattern of [Gang of Four (GOF) Design Patterns in .Net](http://www.dotnettricks.com/learn/designpatterns/gang-of-four-gof-design-patterns-in-net). It is used to builds a complex object by using a step by step approach. Builder interface defines the steps to build the final object. This builder is independent from the objects creation process. A class that is known as Director, controls the object creation process.

Moreover, builder pattern describes a way to separate an object from its construction. The same construction method can create different representation of the object.

***Builder Pattern - UML Diagram & Implementation***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/builder.png | The classes, interfaces and objects in the above UML class diagram are as follows:   1. **Builder** This is an interface which is used to define all the steps to create a product 2. **ConcreteBuilder** This is a class which implements the Builder interface to create complex product. 3. **Product** This is a class which defines the parts of the complex object which are to be generated by the builder pattern. 4. **Director** This is a class which is used to construct an object using the Builder interface. |

C# - Implementation Code

public interface IBuilderv{ void BuildPart1(); void BuildPart2(); void BuildPart3(); Product GetProduct();}

public class ConcreteBuilder : IBuilder

{

private Product \_product = new Product(); public void BuildPart1() { \_product.Part1 = "Part 1"; } public void BuildPart2() { \_product.Part2 = "Part 2"; }

public void BuildPart3() { \_product.Part3 = "Part 3"; } public Product GetProduct() { return \_product; }

}

public class Product{ public string Part1 { get; set; } public string Part2 { get; set; } public string Part3 { get; set; }}

public class Director { public void Construct(IBuilder IBuilder) { IBuilder.BuildPart1(); IBuilder.BuildPart2(); IBuilder.BuildPart3(); }}

***Builder Pattern - Example***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/builder1.png | The classes, interfaces and objects in the above class diagram can be identified as follows:   1. **IVehicleBuilder** - Builder interface 2. **HeroBuilder & HondaBuilder**- Concrete Builder 3. **Vehicle**- Product 4. **Vehicle Creator**- Director |

*/// The 'Builder' interface*

public interface IVehicleBuilder{ void SetModel(); void SetEngine(); void SetTransmission(); void SetBody(); void SetAccessories(); Vehicle GetVehicle();}

public class HeroBuilder : IVehicleBuilder */// The 'ConcreteBuilder1' class*

{

Vehicle obj = new Vehicle(); public void SetModel() { obj.Model = "Hero";} public void SetEngine() { obj.Engine = "4 Stroke"; }

public void SetTransmission() { obj.Transmission = "120 km/hr"; } public void SetBody() { obj.Body = "Plastic"; }

public void SetAccessories() { obj.Accessories.Add("Seat Cover"); obj.Accessories.Add("Rear Mirror");}

public Vehicle GetVehicle() { return obj; }

}

public class HondaBuilder : IVehicleBuilder */// The 'ConcreteBuilder2' class*

{

Vehicle obj = new Vehicle(); public void SetModel() { obj.Model = "Honda"; } public void SetEngine() { obj.Engine = "4 Stroke"; }

  public void SetTransmission() { obj.Transmission = "125 Km/hr"; } public void SetBody() { obj.Body = "Plastic"; }

public void SetAccessories() { obj.Accessories.Add("Seat Cover"); obj.Accessories.Add("Rear Mirror"); obj.Accessories.Add("Helmet"); }

  public Vehicle GetVehicle() { return obj; }

}

public class Vehicle */// The 'Product' class*

{

public string Model { get; set; } public string Engine { get; set; } public string Transmission { get; set; } public string Body { get; set; }

public List<string> Accessories {get; set; } public Vehicle() { Accessories = new List<string>(); }

  public void ShowInfo()

{ Console.WriteLine("Model: {0}", Model); Console.WriteLine("Engine: {0}", Engine); Console.WriteLine("Body: {0}", Body);

Console.WriteLine("Transmission: {0}", Transmission); Console.WriteLine("Accessories:");

foreach (var accessory in Accessories) { Console.WriteLine("\t{0}", accessory); }

}

}

public class Obj

{

private readonly IVehicleBuilder obj;

public Obj(IVehicleBuilder builder) { obj = builder; }

  public void CreateVehicle() { obj.SetModel(); obj.SetEngine(); obj.SetBody(); obj.SetTransmission(); obj.SetAccessories(); }

  public Vehicle GetVehicle() { return obj.GetVehicle(); }

}

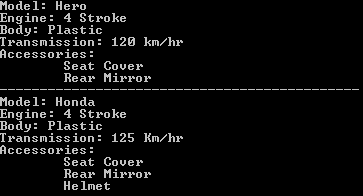
static void Main(string[] args)

{ var obj = new Obj(new HeroBuilder()); obj.CreateVehicle(); var vehicle = obj.GetVehicle(); vehicle.ShowInfo();

obj = new Obj(new HondaBuilder()); obj.CreateVehicle(); vehicle = obj.GetVehicle(); vehicle.ShowInfo();

 }

***Builder Pattern Demo - Output***



***When to use it?***

1. Need to create an object in several steps (a step by step approach).
2. The creation of objects should be independent from the way the object's parts are assembled.
3. Runtime control over the creation process is required.

***Prototype Design Pattern***

Prototype pattern falls under Creational Pattern. It provides an interface for creating parts of a product. Prototype pattern is used to create a duplicate object or clone of the current object to enhance performance. This pattern is used when creation of object is costly or complex. **For Example:** An object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

***Prototype Pattern - UML Diagram & Implementation***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/prototype.png | The classes, interfaces and objects in the above UML class diagram are as follows:   1. **Prototype** This is an interface which is used for the types of object that can be cloned itself. 2. **ConcretePrototype** This is a class which implements the Prototype interface for cloning itself. |

public interface Prototype { Prototype Clone();}

*// Shallow Copy: only top-level objects are duplicated // Deep Copy: all objects are duplicated*  *//return (Prototype)this.Clone();*

public class ConcretePrototypeA : Prototype{ public Prototype Clone() {return (Prototype) MemberwiseClone(); }}

public class ConcretePrototypeB : Prototype { public Prototype Clone() { return (Prototype)MemberwiseClone();}}

## Prototype Pattern - Example

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/prototype1.png | The classes, interfaces and objects in the above class diagram can be identified as follows:   1. **IEmployee** - Prototype interface 2. **Developer & Typist**- Concrete Prototype |

public interface IEmployee { IEmployee Clone(); string GetDetails();} */// The 'Prototype' interface*

public class Developer : IEmployee */// A 'ConcretePrototype' class*

{ public int WordsPerMinute { get; set; } public string Name { get; set; } public string Role { get; set; }

public string PreferredLanguage { get; set; } public IEmployee Clone() {return (IEmployee)MemberwiseClone(); }

public string GetDetails() { return string.Format("{0} - {1} - {2}", Name, Role, PreferredLanguage); }

}

public class Typist : IEmployee */// A 'ConcretePrototype' class*

{ public int WordsPerMinute { get; set; } public string Name { get; set; } public string Role { get; set; }

  public IEmployee Clone() { return (IEmployee)MemberwiseClone(); }

  public string GetDetails() { return string.Format("{0} - {1} - {2}wpm", Name, Role, WordsPerMinute); }

}

static void Main(string[] args)

{

Developer dev = new Developer(); dev.Name = "Rahul"; dev.Role = "Team Leader"; dev.PreferredLanguage = "C#";

  Developer devCopy = (Developer)dev.Clone(); devCopy.Name = "Arif"; *//Not mention Role and PreferredLanguage, it will copy above*

  Console.WriteLine(dev.GetDetails()); Console.WriteLine(devCopy.GetDetails());

Typist typist = new Typist(); typist.Name = "Monu"; typist.Role = "Typist"; typist.WordsPerMinute = 120;

  Typist typistCopy = (Typist)typist.Clone(); typistCopy.Name = "Sahil"; typistCopy.WordsPerMinute = 115;*//Not mention Role, it will copy above*

  Console.WriteLine(typist.GetDetails()); Console.WriteLine(typistCopy.GetDetails());

}

***Prototype Pattern Demo - Output***

http://www.dotnettricks.com/img/designpatterns/Prototype2.png

***When to use it?***

1. The creation of each object is costly or complex.
2. A limited number of state combinations exist in an object.

***Adapter Design Pattern - C#***

* Adapter pattern falls under Structural Pattern. The Adapter pattern allows a system to use classes of another system that is incompatible with it. It is especially used for toolkits and libraries. Adapter pattern acts as a bridge between two incompatible interfaces. This pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces.
* **For Example:** A card reader acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

***Adapter Pattern - UML Diagram & Implementation***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/adapter.png | The classes, interfaces and objects in the above UML class diagram are as follows:   1. **ITarget** This is an interface which is used by the client to achieve its functionality/request. 2. **Adapter** This is a class which implements the ITarget interface and inherits the Adaptee class. It is responsible for communication between 3. **Adaptee** This is a class which have the functionality, required by the client. However, its interface is not compatible with the client. 4. **Client** This is a class which interact with a type that implements the ITarget interface. However, the communication class called adaptee, is not compatible with the client |

public class Client { private ITarget target; public Client(ITarget target) { this.target = target; }  public void MakeRequest() { target.MethodA(); }}

public interface ITarget { void MethodA();}

public class Adapter : Adaptee, ITarget{ public void MethodA() { MethodB(); }}

public class Adaptee { public void MethodB() { Console.WriteLine("MethodB() is called"); }}

***Adapter Pattern - Example***

|  |  |
| --- | --- |
| http://www.dotnettricks.com/img/designpatterns/adapter1.png | The classes, interfaces and objects in the above class diagram can be identified as follows:   1. **ITraget** - Target interface 2. **Employee Adapter**- Adapter Class 3. **HR System**- Adaptee Class 4. **ThirdPartyBillingSystem**- Client |

***C# - Sample Code***

public class ThirdPartyBillingSystem */// The 'Client' class*

{

private ITarget employeeSource; public ThirdPartyBillingSystem(ITarget employeeSource) { this.employeeSource = employeeSource; }

public void ShowEmployeeList() { List<string> employee = employeeSource.GetEmployeeList();

*//To DO: Implement you business logic*

Console.WriteLine("######### Employee List ##########"); foreach (var item in employee) { Console.Write(item); }

}

}

public interface ITarget { List<string> GetEmployeeList();} */// The 'ITarget' interface*

public class HRSystem { public string[][] GetEmployees() { string[][] employees = new string[4][];

employees[0] = new string[] { "100", "Deepak", "Team Leader" }; employees[1] = new string[] { "101", "Rohit", "Developer" };

employees[2] = new string[] { "102", "Gautam", "Developer" }; employees[3] = new string[] { "103", "Dev", "Tester" }; return employees;

}}

public class EmployeeAdapter : HRSystem, ITarget

{

public List<string> GetEmployeeList()

{

List<string> employeeList = new List<string>();

string[][] employees = GetEmployees();

foreach (string[] employee in employees)

{

employeeList.Add(employee[0]);

employeeList.Add(",");

employeeList.Add(employee[1]);

employeeList.Add(",");

employeeList.Add(employee[2]);

employeeList.Add("\n");

}

return employeeList;

}

}

*///*

*/// Adapter Design Pattern Demo*

*///*

class Program

{

static void Main(string[] args)

{

ITarget Itarget = new EmployeeAdapter();

ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);

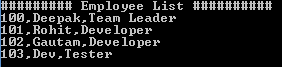
client.ShowEmployeeList();

Console.ReadKey();

}

}

***Adapter Pattern Demo - Output***



***When to use it?***

1. Allow a system to use classes of another system that is incompatible with it.
2. Allow communication between new and already existing system which are independent to each other
3. Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter are best example of Adapter Pattern.

***Note***

1. Internally, Adapter use Factory design pattern for creating objects. But it can also use Builder design pattern and prototype design pattern for creating product. It completely depends upon your implementation for creating products.
2. Adapter can be used as an alternative to Facade to hide platform-specific classes.
3. When Adapter, Builder, and Prototype define a factory for creating the products, we should consider the following points :
   1. Adapter use the factory for creating objects of several classes.
   2. Builder use the factory for creating a complex product by using simple objects and a step by step approach.
   3. Prototype use the factory for building a product by copying an existing product.

***Bridge Design Pattern - C#***

Bridge pattern falls under Structural Pattern of [Gang of Four (GOF) Design Patterns in .Net](http://www.dotnettricks.com/learn/designpatterns/gang-of-four-gof-design-patterns-in-net). All we know, Inheritance is a way to specify different implementations of an abstraction. But in this way, implementations are tightly bound to the abstraction and can not be modified independently. The Bridge pattern provides an alternative to inheritance when there are more than one version of an abstraction. In this article, I would like share what is bridge pattern and how is it work?

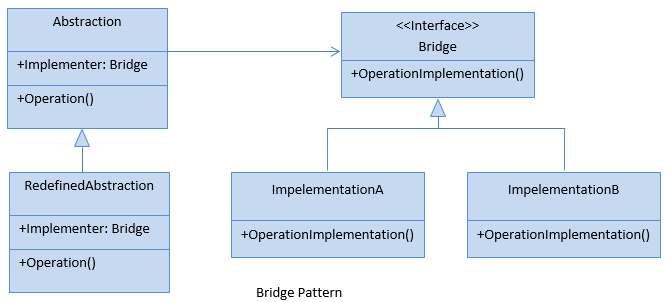
***What is Bridge Pattern***

Bridge pattern is used to separate an abstraction from its implementation so that both can be modified independently.

This pattern involves an interface which acts as a bridge between the abstraction class and implementer classes and also makes the functionality of implementer class independent from the abstraction class. Both types of classes can be modified without affecting to each other.

## Bridge Pattern - UML Diagram & Implementation

The UML class diagram for the implementation of the bridge design pattern is given below:



The classes, interfaces and objects in the above UML class diagram are as follows:

### Abstraction

This is an abstract class and containing members that define an abstract business object and its functionality. It contains a reference to an object of type Bridge. It can also acts as the base class for other abstractions.

### Redefined Abstraction

This is a class which inherits from the Abstraction class. It extends the interface defined by Abstraction class.

### Bridge

This is an interface which acts as a bridge between the abstraction class and implementer classes and also makes the functionality of implementer class independent from the abstraction class.

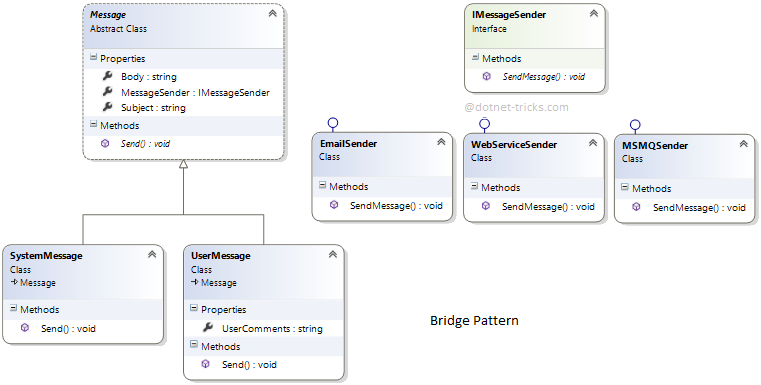
### ImplementationA & ImplementationB

These are classes which implement the Bridge interface and also provide the implementation details for the associated Abstraction class.

***C# - Implementation Code***

1. public abstract class Abstraction
2. {
3. public Bridge Implementer { get; set; }
5. public virtual void Operation()
6. {
7. Console.WriteLine("ImplementationBase:Operation()");
8. Implementer.OperationImplementation();
9. }
10. }
12. public class RefinedAbstraction : Abstraction
13. {
14. public override void Operation()
15. {
16. Console.WriteLine("RefinedAbstraction:Operation()");
17. Implementer.OperationImplementation();
18. }
19. }
21. public interface Bridge
22. {
23. void OperationImplementation();
24. }
26. public class ImplementationA : Bridge
27. {
28. public void OperationImplementation()
29. {
30. Console.WriteLine("ImplementationA:OperationImplementation()");
31. }
32. }
34. public class ImplementationB : Bridge
35. {
36. public void OperationImplementation()
37. {
38. Console.WriteLine("ImplementationB:OperationImplementation()");
39. }
40. }

## Bridge Pattern - Example



### Who is what?

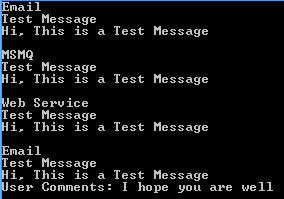
The classes, interfaces and objects in the above class diagram can be identified as follows:

1. **Message** - Abstraction Class.
2. **SystemMessage & UserMessage**- Redefined Abstraction Classes.
3. **IMessageSender**- Bridge Interface.
4. **EmailSender, WebServiceSender & MSMQ Sender**- ConcreteImplementation class which implements the IMessageSender interface.

### C# - Sample Code

1. */// The 'Abstraction' class*
2. public abstract class Message
3. {
4. public IMessageSender MessageSender { get; set; }
5. public string Subject { get; set; }
6. public string Body { get; set; }
7. public abstract void Send();
8. }
10. */// The 'RefinedAbstraction' class*
11. public class SystemMessage : Message
12. {
13. public override void Send()
14. {
15. MessageSender.SendMessage(Subject, Body);
16. }
17. }
19. */// The 'RefinedAbstraction' class*
20. public class UserMessage : Message
21. {
22. public string UserComments { get; set; }
24. public override void Send()
25. {
26. string fullBody = string.Format("{0}\nUser Comments: {1}", Body, UserComments);
27. MessageSender.SendMessage(Subject, fullBody);
28. }
29. }
31. */// The 'Bridge/Implementor' interface*
32. public interface IMessageSender
33. {
34. void SendMessage(string subject, string body);
35. }
37. */// The 'ConcreteImplementor' class*
38. public class EmailSender : IMessageSender
39. {
40. public void SendMessage(string subject, string body)
41. {
42. Console.WriteLine("Email\n{0}\n{1}\n", subject, body);
43. }
44. }
46. */// The 'ConcreteImplementor' class*
47. public class MSMQSender : IMessageSender
48. {
49. public void SendMessage(string subject, string body)
50. {
51. Console.WriteLine("MSMQ\n{0}\n{1}\n", subject, body);
52. }
53. }
55. */// The 'ConcreteImplementor' class*
56. public class WebServiceSender : IMessageSender
57. {
58. public void SendMessage(string subject, string body)
59. {
60. Console.WriteLine("Web Service\n{0}\n{1}\n", subject, body);
61. }
62. }
64. */// Bridge Design Pattern Demo*
65. class Program
66. {
67. static void Main(string[] args)
68. {
69. IMessageSender email = new EmailSender();
70. IMessageSender queue = new MSMQSender();
71. IMessageSender web = new WebServiceSender();
73. Message message = new SystemMessage();
74. message.Subject = "Test Message";
75. message.Body = "Hi, This is a Test Message";
77. message.MessageSender = email;
78. message.Send();
80. message.MessageSender = queue;
81. message.Send();
83. message.MessageSender = web;
84. message.Send();
86. UserMessage usermsg = new UserMessage();
87. usermsg.Subject = "Test Message";
88. usermsg.Body = "Hi, This is a Test Message";
89. usermsg.UserComments = "I hope you are well";
91. usermsg.MessageSender = email;
92. usermsg.Send();
94. Console.ReadKey();
95. }
96. }

### Bridge Pattern Demo - Output



## When to use it?

1. Abstractions and implementations should be modified independently.
2. Changes in the implementation of an abstraction should have no impact on clients.
3. The Bridge pattern is used when a new version of a software or system is brought out, but the older version of the software still running for its existing client. There is no need to change the client code, but the client need to choose which version he wants to use.

#### Note

Bridge pattern has nearly the same structure as the Adapter Pattern. But it is used when designing new systems instead of the Adapter pattern which is used with already existing systems.

##### What do you think?

I hope you will enjoy the bridge Pattern while designing your software. I would like to have feedback from my blog readers. Your valuable feedback, question, or comments about this article are always welcome.

***Composite Design Pattern - C#***

Composite pattern falls under Structural Pattern of [Gang of Four (GOF) Design Patterns in .Net](http://www.dotnettricks.com/learn/designpatterns/gang-of-four-gof-design-patterns-in-net). Composite Pattern is used to arrange structured hierarchies. In this article, I would like share what is composite pattern and how is it work?

## What is Composite Pattern

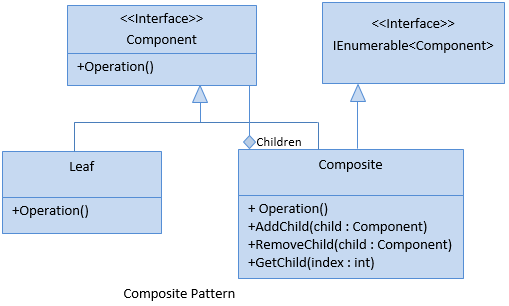
Composite pattern is used to separate an abstraction from its implementation so that both can be modified independently.

Composite pattern is used when we need to treat a group of objects and a single object in the same way. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchies.

This pattern creates a class contains group of its own objects. This class provides ways to modify its group of same objects.

## Composite Pattern - UML Diagram & Implementation

The UML class diagram for the implementation of the composite design pattern is given below:



The classes, interfaces and objects in the above UML class diagram are as follows:

### Component

This is an abstract class containing members that will be implemented by all object in the hierarchy. It acts as the base class for all the objects within the hierarchy

### Composite

This is a class which includes Add,Remove,Find and Get methods to do operations on child components.

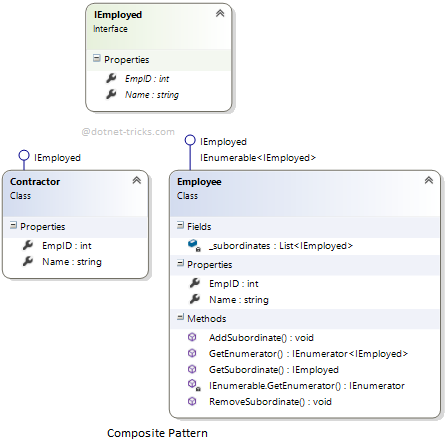
### Leaf

This is a class which is used to define leaf components within the tree structure means these cannot have children.

### C# - Implementation Code

1. public interface Component
2. {
3. void Operation();
4. }
6. public class Composite : Component, IEnumerable
7. {
8. private List \_children = new List();
9. public void AddChild(Component child)
10. {
11. \_children.Add(child);
12. }
13. public void RemoveChild(Component child)
14. {
15. \_children.Remove(child);
16. }
17. public Component GetChild(int index)
18. {
19. return \_children[index];
20. }
21. public void Operation()
22. {
23. string message = string.Format("Composite with {0} child(ren)", \_children.Count);
24. Console.WriteLine(message);
25. }
26. public IEnumerator GetEnumerator()
27. {
28. foreach (Component child in \_children)
29. yield return child;
30. }
31. IEnumerator IEnumerable.GetEnumerator()
32. {
33. return GetEnumerator();
34. }
35. }
36. public class Leaf : Component
37. {
38. public void Operation()
39. {
40. Console.WriteLine("Leaf");
41. }
42. }

## Composite Pattern - Example



### Who is what?

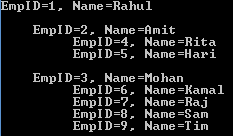
The classes, interfaces and objects in the above class diagram can be identified as follows:

1. **IEmployed** - Component Interface.
2. **Employee**- Composite Class.
3. **Contractor**- Leaf Class.

### C# - Sample Code

1. */// The 'Component' Treenode*
2. public interface IEmployed
3. {
4. int EmpID { get; set; }
5. string Name { get; set; }
6. }
8. */// The 'Composite' class*
9. public class Employee : IEmployed, IEnumerable<IEmployed>
10. {
11. private List<IEmployed> \_subordinates = new List<IEmployed>();
13. public int EmpID { get; set; }
14. public string Name { get; set; }
16. public void AddSubordinate(IEmployed subordinate)
17. {
18. \_subordinates.Add(subordinate);
19. }
21. public void RemoveSubordinate(IEmployed subordinate)
22. {
23. \_subordinates.Remove(subordinate);
24. }
26. public IEmployed GetSubordinate(int index)
27. {
28. return \_subordinates[index];
29. }
31. public IEnumerator<IEmployed> GetEnumerator()
32. {
33. foreach (IEmployed subordinate in \_subordinates)
34. {
35. yield return subordinate;
36. }
37. }
39. IEnumerator IEnumerable.GetEnumerator()
40. {
41. return GetEnumerator();
42. }
43. }
45. */// The 'Leaf' class*
46. public class Contractor : IEmployed
47. {
48. public int EmpID { get; set; }
49. public string Name { get; set; }
50. }
52. class Program
53. {
54. static void Main(string[] args)
55. {
56. Employee Rahul = new Employee { EmpID = 1, Name = "Rahul" };
58. Employee Amit = new Employee { EmpID = 2, Name = "Amit" };
59. Employee Mohan = new Employee { EmpID = 3, Name = "Mohan" };
61. Rahul.AddSubordinate(Amit);
62. Rahul.AddSubordinate(Mohan);
64. Employee Rita = new Employee { EmpID = 4, Name = "Rita" };
65. Employee Hari = new Employee { EmpID = 5, Name = "Hari" };
67. Amit.AddSubordinate(Rita);
68. Amit.AddSubordinate(Hari);
70. Employee Kamal = new Employee { EmpID = 6, Name = "Kamal" };
71. Employee Raj = new Employee { EmpID = 7, Name = "Raj" };
73. Contractor Sam = new Contractor { EmpID = 8, Name = "Sam" };
74. Contractor tim = new Contractor { EmpID = 9, Name = "Tim" };
76. Mohan.AddSubordinate(Kamal);
77. Mohan.AddSubordinate(Raj);
78. Mohan.AddSubordinate(Sam);
79. Mohan.AddSubordinate(tim);
81. Console.WriteLine("EmpID={0}, Name={1}", Rahul.EmpID, Rahul.Name);
83. foreach (Employee manager in Rahul)
84. {
85. Console.WriteLine("\n EmpID={0}, Name={1}", manager.EmpID, manager.Name);
87. foreach (var employee in manager)
88. {
89. Console.WriteLine(" \t EmpID={0}, Name={1}", employee.EmpID, employee.Name);
90. }
91. }
92. Console.ReadKey();
93. }
94. }

### composite Pattern Demo - Output



## When to use it?

1. Hierarchical representations of objects are required.
2. A single object and a group of objects should be treated in the same way.
3. The Composite pattern is used when data is organized in a tree structure (for example directories in a computer).

##### What do you think?

I hope you will enjoy the composite Pattern while designing your software. I would like to have feedback from my blog readers. Your valuable feedback, question, or comments about this article are always welcome.