Software design principles are a set of guidelines that helps developers to make a good system design. The most important principle is SOLID principles and interactions of objects. And Design patterns provide solutions to common problems which occur in software design. Design patterns are about reusable designs and interactions of objects.

Types of Design Patterns:

These can be organized in 4 separate pattern groups depending on the nature of the design problem they intend to solve.

* Gang of Four Patterns
* Enterprise Patterns
* SOA and Messaging Patterns
* Model-View Patterns

The 23 Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral

**Creational Design patterns:**

These patterns deal with the process of objects creation in such a way that they can be decoupled from their implementing system. This provides more flexibility in deciding which objects need to be created for a given use case/ scenario. There are as follows:

[Factory Method](http://www.dotnettricks.com/learn/designpatterns/factory-method-design-pattern-dotnet) : Create instances of derived classes

[Abstract Factory](http://www.dotnettricks.com/learn/designpatterns/abstract-factory-design-pattern-dotnet): Create instances of several classes belonging to different families

[Builder](http://www.dotnettricks.com/learn/designpatterns/builder-design-pattern-dotnet) : Separates an object construction from its representation

[Prototype](http://www.dotnettricks.com/learn/designpatterns/prototype-design-pattern-dotnet") : Create a duplicate object or clone of the object

[Singleton](http://www.dotnettricks.com/learn/designpatterns/singleton-design-pattern-dotnet) : Ensures that a class can has only one instance

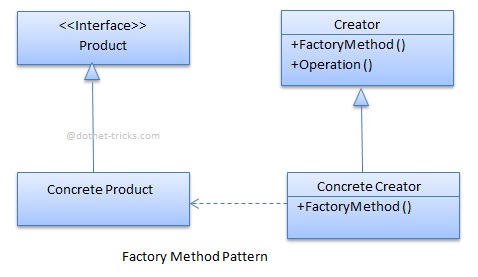
**Factory:**

## What is Factory Method Pattern?

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 sub classes.

## Factory Method Pattern - UML Diagram & Implementation

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



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

### Product This is an interface for creating the objects.

### ConcreteProduct This is a class which implements the Product interface.

### Creator This is an abstract class and declares the factory method, which returns an object of type Product.

### 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 Standard Structure:

|  |
| --- |
| public interface IProduct  {  void Display();  }  public class ConcreteProductA : IProduct  {  public void Display()  {  Console.WriteLine("ConcreteProductA");  }  }  public class ConcreteProductB : IProduct  {  public void Display()  {  Console.WriteLine("ConcreteProductB");  }  }  public class ConcreteProductC : IProduct  {  public void Display()  {  Console.WriteLine("ConcreteProductC");  }  }  public abstract class Creator  {  public abstract IProduct FactoryMethod(string type);  }  public class ConcreteCreator : Creator  {  public override IProduct FactoryMethod(string type)  {  IProduct product = null;  if (type.Equals("ConcreteProductA"))  {  product = new ConcreteProductA();  }  else if (type.Equals("ConcreteProductB"))  {  product = new ConcreteProductB();  }  else if (type.Equals("ConcreteProductC"))  {  product = new ConcreteProductC();  }  return product;  }  }  static void Main(string[] args)  {  //Creating Objects by Factory Pattern  Creator creator = new ConcreteCreator();  IProduct productA = creator.FactoryMethod("ConcreteProductA");  productA.Display();  IProduct productB = creator.FactoryMethod("ConcreteProductB");  productB.Display();  IProduct productC = creator.FactoryMethod("ConcreteProductC");  productC.Display();  Console.ReadLine();  } |

**RealWorld example**

|  |
| --- |
| /// <summary>  /// The 'Product' interface  /// </summary>  public interface IFactory  {  void Drive(int miles);  }  /// <summary>  /// A 'ConcreteProduct' class  /// </summary>  public class Scooter : IFactory  {  public void Drive(int miles)  {  Console.WriteLine("Drive the Scooter : " + miles.ToString() + "km");  }  }  /// <summary>  /// A 'ConcreteProduct' class  /// </summary>  public class Bike : IFactory  {  public void Drive(int miles)  {  Console.WriteLine("Drive the Bike : " + miles.ToString() + "km");  }  }  /// <summary>  /// The Creator Abstract Class  /// </summary>  public abstract class VehicleFactory  {  public abstract IFactory GetVehicle(string Vehicle);  }  /// <summary>  /// A 'ConcreteCreator' class  /// </summary>  public class ConcreteVehicleFactory : VehicleFactory  {  public override IFactory GetVehicle(string Vehicle)  {  switch (Vehicle)  {  case "Scooter":  return new Scooter();  case "Bike":  return new Bike();  default:  throw new ApplicationException(string.Format("Vehicle '{0}' cannot be created", Vehicle));  }  }  }  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);  Console.ReadLine();  } |

**Abstract Factory:**

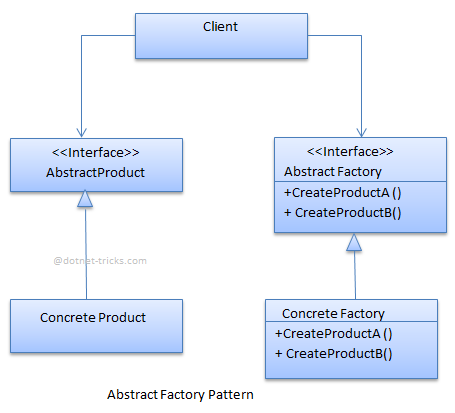
Abstract Factory method pattern falls under Creational Pattern of [Gang of Four (GOF) Design Patterns in .Net](http://www.dotnettricks.com/learn/designpatterns/gang-of-four-gof-design-patterns-in-net). It is used to create a set of related objects, or dependent objects. Internally, Abstract Factory use 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. In this article, I would like share what is abstract factory pattern and how is it work?

## What is Abstract Factory 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.

## Abstract Factory Pattern - UML Diagram & Implementation

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



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

### AbstractFactory This is an interface which is used to create abstract product

### ConcreteFactor yThis is a class which implements the AbstractFactory interface to create concrete products.

### AbstractProduct This is an interface which declares a type of product.

### ConcreteProduct This is a class which implements the AbstractProduct interface to create product.

### Client This is a class which use AbstractFactory and AbstractProduct interfaces to create a family of related objects.

### C# - Implementation Code standard structure

|  |
| --- |
| public abstract class AbstractProductA  {  public virtual void DisplayProductADetails()  {  Console.WriteLine("Default Details of Product A");  }  }  public abstract class AbstractProductB  {  public virtual void DisplayProductBDetails()  {  Console.WriteLine("Default Details of Product B");  }  }  public class ProductA1 : AbstractProductA  {  public override void DisplayProductADetails()  {  Console.WriteLine("In Product A1");  }  }  public class ProductA2 : AbstractProductA  {  }  public class ProductB1 : AbstractProductB  {  public override void DisplayProductBDetails()  {  Console.WriteLine("In Product B1");  }  }  public class ProductB2 : AbstractProductB  {  }  public interface IAbstractFactory  {  AbstractProductA CreateProductA();  AbstractProductB CreateProductB();  }  public class ConcreteFactory1 : IAbstractFactory  {  public AbstractProductA CreateProductA()  {  return new ProductA1();  }  public AbstractProductB CreateProductB()  {  return new ProductB1();  }  }  public class ConcreteFactory2 : IAbstractFactory  {  public AbstractProductA CreateProductA()  {  return new ProductA2();  }  public AbstractProductB CreateProductB()  {  return new ProductB2();  }  }  public class Client  {  public AbstractProductA \_abstractProductA;  public AbstractProductB \_abstractProductB;  public Client(IAbstractFactory \_abstractFactory)  {  \_abstractProductA = \_abstractFactory.CreateProductA();  \_abstractProductB = \_abstractFactory.CreateProductB();  }  }  static void Main(string[] args)  {  IAbstractFactory \_abstractFactory1 = new ConcreteFactory1();  Client \_client1 = new Client(\_abstractFactory1);  AbstractProductA \_productA1 = \_client1.\_abstractProductA;  \_productA1.DisplayProductADetails();  AbstractProductB \_productB1 = \_client1.\_abstractProductB;  \_productB1.DisplayProductBDetails();  IAbstractFactory \_abstractFactory2 = new ConcreteFactory2();  Client \_client2 = new Client(\_abstractFactory2);  AbstractProductA \_productA2 = \_client2.\_abstractProductA;  \_productA2.DisplayProductADetails();  AbstractProductB \_productB2 = \_client2.\_abstractProductB;  \_productB2.DisplayProductBDetails();  Console.ReadLine();  } |

**Realworld example:**

|  |
| --- |
| //Abstract types  public interface IBike  {  string Name { get; set; }  }  public interface IScooter  {  string Name { get; set; }  }  //Concrete types/products  public class SportsBike : IBike  {  public string Name { get; set; }  public SportsBike(string name)  {  Name = name;  }  }  public class RegularBike : IBike  {  public string Name { get; set; }  public RegularBike(string name)  {  Name = name;  }  }  public class Scooty : IScooter  {  public string Name { get; set; }  public Scooty(string name)  {  Name = name;  }  }  public class RegularScooter : IScooter  {  public string Name { get; set; }  public RegularScooter(string name)  {  Name = name;  }  }  //abstract factory  public interface IVehicleFactory  {  IBike GetBike(string type);  IScooter GetScooter(string type);  }  //Concrete Factories  public class HondaFactory : IVehicleFactory  {  public IBike GetBike(string type)  {  string Name = "Honda " + type + " bike";  if (type.Equals("Sports"))  return new SportsBike(Name);  if (type.Equals("Regular"))  return new RegularBike(Name);  throw new ApplicationException(string.Format("Vehicle '{0}' cannot be created", type));  }  public IScooter GetScooter(string type)  {  string Name = "Honda " + type + " scooter";  if (type.Equals("Sports"))  return new Scooty(Name + " Scooty");  if (type.Equals("Regular"))  return new RegularScooter(Name);  throw new ApplicationException(string.Format("Vehicle '{0}' cannot be created", type));  }  }  public class HeroFactory : IVehicleFactory  {  public IBike GetBike(string type)  {  string Name = "Hero " + type + " bike";  if (type.Equals("Sports"))  return new SportsBike(Name);  if (type.Equals("Regular"))  return new RegularBike(Name);  throw new ApplicationException(string.Format("Vehicle '{0}' cannot be created", type));  }  public IScooter GetScooter(string type)  {  string Name = "Hero " + type + " scooter";  if (type.Equals("Sports"))  return new Scooty(Name + " Scooty");  if (type.Equals("Regular"))  return new RegularScooter(Name);  throw new ApplicationException(string.Format("Vehicle '{0}' cannot be created", type));  }  }  public class VehicleClient  {  IBike \_bike;  IScooter \_scooter;  public VehicleClient(IVehicleFactory factory, string type)  {  \_bike = factory.GetBike(type);  \_scooter = factory.GetScooter(type);  }  public void DisplayProductDetails()  {  Console.WriteLine("Bike : " + \_bike.Name);  Console.WriteLine("Scooter : " + \_scooter.Name);  }  }  static void Main(string[] args)  {  IVehicleFactory hondaFactory = new HondaFactory();  VehicleClient hondaClient = new VehicleClient(hondaFactory, "Regular");  hondaClient.DisplayProductDetails();  Console.WriteLine();  IVehicleFactory heroFactory = new HeroFactory();  VehicleClient heroClient = new VehicleClient(heroFactory, "Sports");  heroClient.DisplayProductDetails();  Console.ReadLine();  } |

**Builder 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. It provides an interface for creating parts of a product.

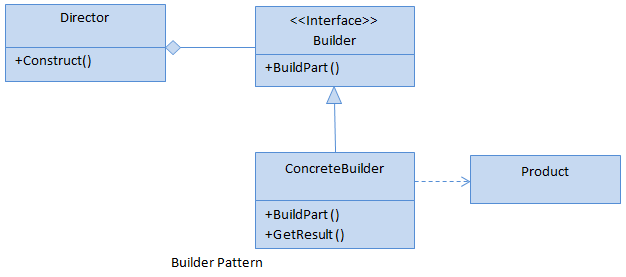
What is Builder Pattern?

Builder pattern 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

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



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

### Builder This is an interface which is used to define all the steps to create a product

### ConcreteBuilder This is a class which implements the Builder interface to create complex product.

### Product This is a class which defines the parts of the complex object which are to be generated by the builder pattern.

### Director This is a class which is used to construct an object using the Builder interface.

### C# - Implementation Code

|  |
| --- |
| public interface IBuilder  {  void BuildPart1();  void BuildPart2();  void BuildPart3();  Product GetProduct();  }  public class ConcreteBuilder : IBuilder  {  private Product \_product = new Product();  public void BuildPart1()  {  \_product.Part1 = "Builded Part 1";  }  public void BuildPart2()  {  \_product.Part2 = "Builded Part 2";  }  public void BuildPart3()  {  \_product.Part3 = "Builded 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 void ShowInfo()  {  Console.WriteLine("Part 1 " + Part1);  Console.WriteLine("Part 2 " + Part2);  Console.WriteLine("Part 3 " + Part3);  }  }  public class Director  {  IBuilder builder;  public Director(IBuilder \_builder)  {  builder = \_builder;  }  public void Construct()  {  builder.BuildPart1();  builder.BuildPart2();  builder.BuildPart3();  }  public Product GetProduct()  {  return builder.GetProduct();  }  }  static void Main(string[] args)  {  var director = new Director(new ConcreteBuilder());  director.Construct();  var product = director.GetProduct();  product.ShowInfo();  Console.ReadLine();  } |

### IMG_257

### **Realworld Example**

### Who is what?

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

**IVehicleBuilder** - Builder interface

**HeroBuilder & HondaBuilder**- Concrete Builder

**Vehicle**- Product

**Vehicle Creator** - Director

C# - Sample Code

|  |
| --- |
| public interface IVehicleBuilder  {  void SetModel();  void SetEngine();  void SetTransmission();  void SetBody();  void SetAccessories();  Vehicle GetVehicle();  }  public class HeroBuilder : IVehicleBuilder  {  Vehicle objVehicle = new Vehicle();  public void SetAccessories()  {  objVehicle.Accessories.Add("Seat Cover");  objVehicle.Accessories.Add("Rear Mirror");  }  public void SetBody()  {  objVehicle.Body = "Plastic";  }  public void SetEngine()  {  objVehicle.Engine = "4 Stroke";  }  public void SetModel()  {  objVehicle.Model = "Hero";  }  public void SetTransmission()  {  objVehicle.Transmission = "120 km/hr";  }  public Vehicle GetVehicle()  {  return objVehicle;  }  }  public class HondaBuilder : IVehicleBuilder  {  Vehicle objVehicle = new Vehicle();  public void SetModel()  {  objVehicle.Model = "Honda";  }  public void SetEngine()  {  objVehicle.Engine = "4 Stroke";  }  public void SetTransmission()  {  objVehicle.Transmission = "125 Km/hr";  }  public void SetBody()  {  objVehicle.Body = "Metal";  }  public void SetAccessories()  {  objVehicle.Accessories.Add("Seat Cover");  objVehicle.Accessories.Add("Rear Mirror");  objVehicle.Accessories.Add("Helmet");  objVehicle.Accessories.Add("Ladies Foot Rest");  }  public Vehicle GetVehicle()  {  return objVehicle;  }  }  public class Vehicle  {  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);  }  }  }  //Director class  public class VehicleCreator  {  IVehicleBuilder builder;  public VehicleCreator(IVehicleBuilder \_builder)  {  builder = \_builder;  }  public void ConstructVehicle()  {  builder.SetAccessories();  builder.SetBody();  builder.SetEngine();  builder.SetModel();  builder.SetTransmission();  }  public Vehicle GetVehicle()  {  return builder.GetVehicle();  }  }  static void Main(string[] args)  {  //Builder Pattern  BuilderPatternRealWorldDemo();  Console.ReadLine();  } |

## When to use it?

## Need to create an object in several steps (a step by step approach).

## The creation of objects should be independent from the way the object's parts are assembled.

**Prototype Pattern**

Prototype 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 used to create a duplicate object or clone of the current object. It provides an interface for creating parts of a product. In this article, I would like share what is Prototype pattern and how is it work?

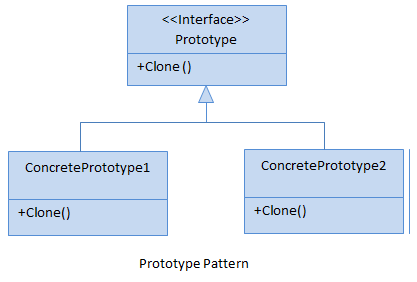
What is Prototype Pattern?

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

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



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

### Prototype This is an interface which is used for the types of object that can be cloned itself.

### ConcretePrototype This is a class which implements the Prototype interface for cloning itself.

### C# - Implementation Code

**public interface Prototype**

**{**

**Prototype Clone();**

**}**

**public class ConcretePrototypeA : Prototype**

**{**

**public Prototype Clone()**

**{**

***// Shallow Copy: only top-level objects are duplicated***

**return (Prototype)MemberwiseClone();**

***// Deep Copy: all objects are duplicated***

***//return (Prototype)this.Clone();***

**}**

**}**

**public class ConcretePrototypeB : Prototype**

**{**

**public Prototype Clone()**

**{**

***// Shallow Copy: only top-level objects are duplicated***

**return (Prototype)MemberwiseClone();**

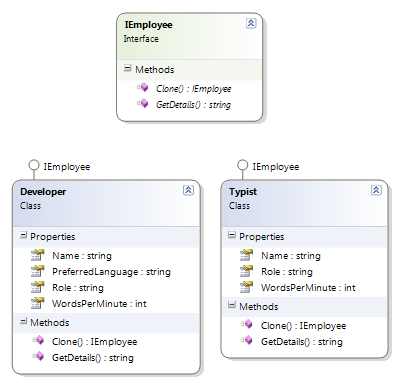
***// Deep Copy: all objects are duplicated***

***//return (Prototype)this.Clone();***

**}**

**}**

## Prototype Pattern - Example



### Who is what?

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

**IEmployee** - Prototype interface

**Developer & Typist**- Concrete Prototype

### C# - Sample Code

***/// <summary>***

***/// The 'Prototype' interface***

***/// </summary>***

**public interface IEmployee**

**{**

**IEmployee Clone();**

**string GetDetails();**

**}**

***/// <summary>***

***/// A 'ConcretePrototype' class***

***/// </summary>***

**public class Developer : IEmployee**

**{**

**public int WordsPerMinute { get; set; }**

**public string Name { get; set; }**

**public string Role { get; set; }**

**public string PreferredLanguage { get; set; }**

**public IEmployee Clone()**

**{**

***// Shallow Copy: only top-level objects are duplicated***

**return (IEmployee)MemberwiseClone();**

***// Deep Copy: all objects are duplicated***

***//return (IEmployee)this.Clone();***

**}**

**public string GetDetails()**

**{**

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

**}**

**}**

***/// <summary>***

***/// A 'ConcretePrototype' class***

***/// </summary>***

**public class Typist : IEmployee**

**{**

**public int WordsPerMinute { get; set; }**

**public string Name { get; set; }**

**public string Role { get; set; }**

**public IEmployee Clone()**

**{**

***// Shallow Copy: only top-level objects are duplicated***

**return (IEmployee)MemberwiseClone();**

***// Deep Copy: all objects are duplicated***

***//return (IEmployee)this.Clone();***

**}**

**public string GetDetails()**

**{**

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

**}**

**}**

***/// <summary>***

***/// Prototype Pattern Demo***

***/// </summary>***

**class Program**

**{**

**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());**

**Console.ReadKey();**

**}**

**}**

### Prototype Pattern Demo - Output

IMG_258

## When to use it?

The creation of each object is costly or complex.