Design Patterns C#

http://www.dofactory.com/net/design-patterns

Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects.

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 (for a complete list see below).

To give you a head start, the C# source code for each pattern is provided in 2 forms:*structural* and *real-world*. Structural code uses type names as defined in the pattern definition and UML diagrams. Real-world code provides real-world programming situations where you may use these patterns.

A third form, *.NET optimized*, demonstrates design patterns that fully exploit built-in .NET 4.5 features, such as, generics, attributes, delegates, reflection, and more. These and much more are available in our .NET Design Pattern Framework 4.5. You can see the Singletonpage for a .NET 4.5 Optimized example.

Creational Patterns	
Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object’s interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Facade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

Behavioral Patterns	
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

Singleton

A class of which only a single instance can exist

Definition

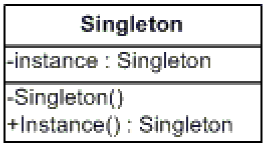
Ensure a class has only one instance and provide a global point of access to it.

Frequency of use:

1 2 3 4 5
🟥🟥🟥🟥🟨

Medium high

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Singleton (LoadBalancer)**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

Structural code in C#

This structural code demonstrates the Singleton pattern which assures only a single instance (the singleton) of the class can be created.

```


using System;

namespace DoFactory.GangOfFour.Singleton.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Singleton Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Constructor is protected -- cannot use new
      Singleton s1 = Singleton.Instance();
      Singleton s2 = Singleton.Instance();

      // Test for same instance
      if (s1 == s2)
      {
        Console.WriteLine("Objects are the same instance");
      }

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Singleton' class
  /// </summary>
  class Singleton
  {
    private static Singleton _instance;

    // Constructor is 'protected'
    protected Singleton()
    {
    }

    public static Singleton Instance()
    {
      // Uses lazy initialization.
      // Note: this is not thread safe.
      if (_instance == null)
      {
        _instance = new Singleton();
      }

      return _instance;
    }
  }
}
```

Output

```
Objects are the same instance
```

Real-world code in C#

This real-world code demonstrates the Singleton pattern as a LoadBalancing object. Only a single instance (the singleton) of the class can be created because servers may dynamically come on- or off-line and every request must go throught the one object that has knowledge about the state of the (web) farm.

```


using System;
using System.Collections.Generic;
using System.Threading;

namespace DoFactory.GangOfFour.Singleton.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Singleton Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
```

```
        static void Main()
        {
            LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
            LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
            LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
            LoadBalancer b4 = LoadBalancer.GetLoadBalancer();

            // Same instance?
            if (b1 == b2 && b2 == b3 && b3 == b4)
            {
                Console.WriteLine("Same instance\n");
            }

            // Load balance 15 server requests
            LoadBalancer balancer = LoadBalancer.GetLoadBalancer();
            for (int i = 0; i < 15; i++)
            {
                string server = balancer.Server;
                Console.WriteLine("Dispatch Request to: " + server);
            }

            // Wait for user
            Console.ReadKey();
```

```
    }
  }

  /// <summary>
  /// The 'Singleton' class
  /// </summary>
  class LoadBalancer
  {
    private static LoadBalancer _instance;
    private List<string> _servers = new List<string>();
    private Random _random = new Random();

    // Lock synchronization object
    private static object syncLock = new object();

    // Constructor (protected)
    protected LoadBalancer()
    {
      // List of available servers
      _servers.Add("ServerI");
      _servers.Add("ServerII");
      _servers.Add("ServerIII");
      _servers.Add("ServerIV");
      _servers.Add("ServerV");
    }

    public static LoadBalancer GetLoadBalancer()
    {
      // Support multithreaded applications through
      // 'Double checked locking' pattern which (once
      // the instance exists) avoids locking each
      // time the method is invoked
      if (_instance == null)
      {
        lock (syncLock)
        {
          if (_instance == null)
          {
            _instance = new LoadBalancer();
          }
        }
      }

      return _instance;
    }

```

```
        // Simple, but effective random load balancer
        public string Server
        {
          get
          {
            int r = _random.Next(_servers.Count);
            return _servers[r].ToString();
          }
        }
      }
    }



```

Output

```
Same instance

ServerIII
ServerII
ServerI
ServerII
ServerI
ServerIII
ServerI
ServerIII
ServerIV
ServerII
ServerII
```

```
ServerIII
ServerIV
ServerII
ServerIV
```

.NET Optimized code in C#

The .NET optimized code demonstrates the same code as above but uses more modern, built-in .NET features.

Here an elegant .NET specific solution is offered. The Singleton pattern simply uses a private constructor and a static readonlyinstance variable that is lazily initialized. Thread safety is guaranteed by the compiler.

```


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Singleton.NETOptimized
{
  /// <summary>
  /// MainApp startup class for .NET optimized
/// Singleton Design Pattern.
/// </summary>
class MainApp
{
  /// <summary>
  /// Entry point into console application.
  /// </summary>
  static void Main()
  {
    LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b4 = LoadBalancer.GetLoadBalancer();

    // Confirm these are the same instance
    if (b1 == b2 && b2 == b3 && b3 == b4)
    {
      Console.WriteLine("Same instance\n");
    }

    // Next, load balance 15 requests for a server
    LoadBalancer balancer = LoadBalancer.GetLoadBalancer();
    for (int i = 0; i < 15; i++)
```

```
        {
          string serverName = balancer.NextServer.Name;
          Console.WriteLine("Dispatch request to: " + serverName);
        }

        // Wait for user
        Console.ReadKey();
      }
    }

    /// <summary>
    /// The 'Singleton' class
    /// </summary>
    sealed class LoadBalancer
    {
      // Static members are 'eagerly initialized', that is,
      // immediately when class is loaded for the first time.
      // .NET guarantees thread safety for static initialization
      private static readonly LoadBalancer _instance =
        new LoadBalancer();

      // Type-safe generic list of servers
      private List<Server> _servers;
      private Random _random = new Random();

      // Note: constructor is 'private'
      private LoadBalancer()
      {
        // Load list of available servers
        _servers = new List<Server>
          {
           new Server{ Name = "ServerI", IP = "120.14.220.18" },
           new Server{ Name = "ServerII", IP = "120.14.220.19" },
           new Server{ Name = "ServerIII", IP = "120.14.220.20" },
           new Server{ Name = "ServerIV", IP = "120.14.220.21" },
           new Server{ Name = "ServerV", IP = "120.14.220.22" },
          };
      }

      public static LoadBalancer GetLoadBalancer()
      {
        return _instance;
      }

      // Simple, but effective load balancer
      public Server NextServer
      {
        get
        {
          int r = _random.Next(_servers.Count);
          return _servers[r];
        }
      }
    }

    /// <summary>
    /// Represents a server machine
    /// </summary>
    class Server
    {
      // Gets or sets server name
      public string Name { get; set; }

      // Gets or sets server IP address
      public string IP { get; set; }
    }
  }


```

Output

```
Same instance

ServerIV
ServerIV
ServerIII
ServerV
ServerII
ServerV
ServerII
ServerII
ServerI
ServerIV
ServerIV
ServerII
ServerI
ServerV
ServerIV
```

Prototype

A fully initialized instance to be copied or cloned

Abstract Factory

Definition

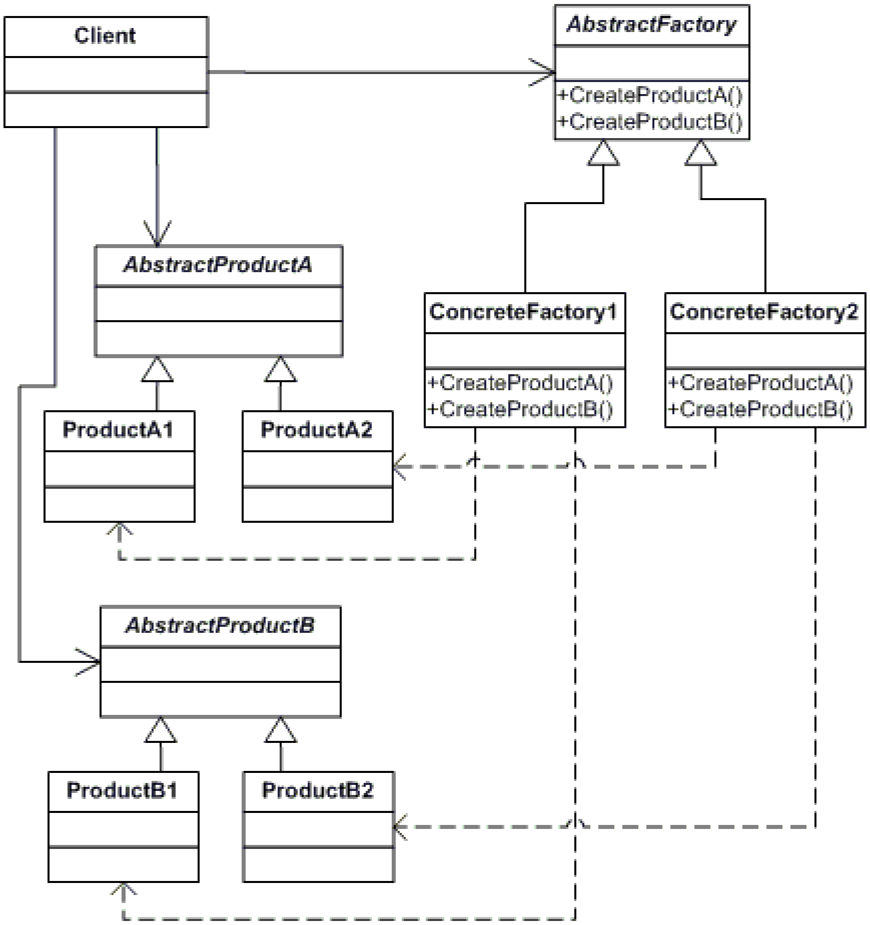
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Frequency of use:

1 2 3 4 5
🟥🟥🟥🟥🟥

High

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **AbstractFactory (ContinentFactory)**
    - declares an interface for operations that create abstract products
- **ConcreteFactory (AfricaFactory, AmericaFactory)**
    - implements the operations to create concrete product objects
- **AbstractProduct (Herbivore, Carnivore)**
    - declares an interface for a type of product object
- **Product (Wildebeest, Lion, Bison, Wolf)**
    - defines a product object to be created by the corresponding concrete factory
    - implements the AbstractProduct interface
- **Client (AnimalWorld)**
    - uses interfaces declared by AbstractFactory and AbstractProduct classes

Structural code in C#

This structural code demonstrates the Abstract Factory pattern creating parallel hierarchies of objects. Object creation has been abstracted and there is no need for hard-coded class names in the client code.

```


using System;

namespace DoFactory.GangOfFour.Abstract.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Abstract factory #1
      AbstractFactory factory1 = new ConcreteFactory1();
      Client client1 = new Client(factory1);
      client1.Run();

      // Abstract factory #2
      AbstractFactory factory2 = new ConcreteFactory2();
      Client client2 = new Client(factory2);
      client2.Run();

      // Wait for user input
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class AbstractFactory
  {
    public abstract AbstractProductA CreateProductA();
    public abstract AbstractProductB CreateProductB();
  }


  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class ConcreteFactory1 : AbstractFactory
  {
    public override AbstractProductA CreateProductA()
    {
      return new ProductA1();
    }
    public override AbstractProductB CreateProductB()
    {
      return new ProductB1();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
  /// </summary>
  class ConcreteFactory2 : AbstractFactory
  {
    public override AbstractProductA CreateProductA()
    {
      return new ProductA2();
    }
    public override AbstractProductB CreateProductB()
    {
      return new ProductB2();
    }
  }

  /// <summary>
  /// The 'AbstractProductA' abstract class
  /// </summary>
  abstract class AbstractProductA
  {
  }

  /// <summary>
  /// The 'AbstractProductB' abstract class
  /// </summary>
  abstract class AbstractProductB
  {
    public abstract void Interact(AbstractProductA a);
  }


  /// <summary>
  /// The 'ProductA1' class
  /// </summary>
  class ProductA1 : AbstractProductA
```

```
  {
  }

  /// <summary>
  /// The 'ProductB1' class
  /// </summary>
  class ProductB1 : AbstractProductB
  {
    public override void Interact(AbstractProductA a)
    {
      Console.WriteLine(this.GetType().Name +
        " interacts with " + a.GetType().Name);
    }
  }

  /// <summary>
  /// The 'ProductA2' class
  /// </summary>
  class ProductA2 : AbstractProductA
  {
  }

  /// <summary>
```

```
/// The 'ProductB2' class
/// </summary>
class ProductB2 : AbstractProductB
{
  public override void Interact(AbstractProductA a)
  {
    Console.WriteLine(this.GetType().Name +
      " interacts with " + a.GetType().Name);
  }
}

/// <summary>
/// The 'Client' class. Interaction environment for the products.
/// </summary>
class Client
{
  private AbstractProductA _abstractProductA;
  private AbstractProductB _abstractProductB;

  // Constructor
  public Client(AbstractFactory factory)
  {
    _abstractProductB = factory.CreateProductB();
```

```
      _abstractProductA = factory.CreateProductA();
    }

    public void Run()
    {
      _abstractProductB.Interact(_abstractProductA);
    }
  }
}



```

Output

```
ProductB1 interacts with ProductA1
ProductB2 interacts with ProductA2
```

Real-world code in C#

This real-world code demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

```


using System;

namespace DoFactory.GangOfFour.Abstract.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Abstract Factory Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create and run the African animal world
      ContinentFactory africa = new AfricaFactory();
      AnimalWorld world = new AnimalWorld(africa);
      world.RunFoodChain();

      // Create and run the American animal world
      ContinentFactory america = new AmericaFactory();
      world = new AnimalWorld(america);
      world.RunFoodChain();

      // Wait for user input
      Console.ReadKey();
    }
  }


  /// <summary>
  /// The 'AbstractFactory' abstract class
  /// </summary>
  abstract class ContinentFactory
  {
    public abstract Herbivore CreateHerbivore();
    public abstract Carnivore CreateCarnivore();
  }

  /// <summary>
  /// The 'ConcreteFactory1' class
  /// </summary>
  class AfricaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
      return new Lion();
    }
  }

  /// <summary>
  /// The 'ConcreteFactory2' class
  /// </summary>
  class AmericaFactory : ContinentFactory
  {
    public override Herbivore CreateHerbivore()
    {
      return new Bison();
    }
    public override Carnivore CreateCarnivore()
    {
      return new Wolf();
    }
  }

  /// <summary>
  /// The 'AbstractProductA' abstract class
  /// </summary>
  abstract class Herbivore
  {
  }

  /// <summary>
  /// The 'AbstractProductB' abstract class
  /// </summary>
  abstract class Carnivore
  {
    public abstract void Eat(Herbivore h);
  }

  /// <summary>
  /// The 'ProductA1' class
  /// </summary>
  class Wildebeest : Herbivore
  {
  }

  /// <summary>
  /// The 'ProductB1' class
  /// </summary>
  class Lion : Carnivore
  {
    public override void Eat(Herbivore h)
    {
      // Eat Wildebeest
      Console.WriteLine(this.GetType().Name +
        " eats " + h.GetType().Name);
    }
  }

  /// <summary>
  /// The 'ProductA2' class
  /// </summary>
  class Bison : Herbivore
  {
  }

  /// <summary>
  /// The 'ProductB2' class
  /// </summary>
  class Wolf : Carnivore
  {
    public override void Eat(Herbivore h)
    {
      // Eat Bison
      Console.WriteLine(this.GetType().Name +
        " eats " + h.GetType().Name);
    }
  }

  /// <summary>
  /// The 'Client' class
  /// </summary>
  class AnimalWorld
  {
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
      _carnivore = factory.CreateCarnivore();
      _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
      _carnivore.Eat(_herbivore);
    }
  }
 }


```

Output

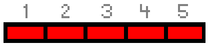
```
Lion eats Wildebeest
Wolf eats Bison
```

Factory Method

Definition

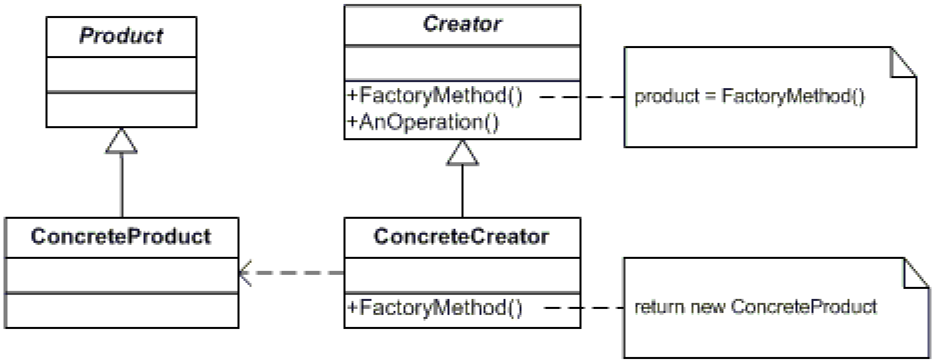
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Frequency of use:



High

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Product (Page)**
    - defines the interface of objects the factory method creates
- **ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)**
    - implements the Product interface
- **Creator (Document)**
    - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
    - may call the factory method to create a Product object.
- **ConcreteCreator (Report, Resume)**
    - overrides the factory method to return an instance of a ConcreteProduct.

Structural code in C#

This structural code demonstrates the Factory method offering great flexibility in creating different objects. The Abstract class may provide a default object, but each subclass can instantiate an extended version of the object.

```


using System;

namespace DoFactory.GangOfFour.Factory.Structural
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Factory Method Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // An array of creators
      Creator[] creators = new Creator[2];

      creators[0] = new ConcreteCreatorA();
      creators[1] = new ConcreteCreatorB();

      // Iterate over creators and create products
      foreach (Creator creator in creators)
      {
        Product product = creator.FactoryMethod();
        Console.WriteLine("Created {0}",
          product.GetType().Name);
      }

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Product' abstract class
  /// </summary>
  abstract class Product
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class ConcreteProductA : Product
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class ConcreteProductB : Product
  {
  }

  /// <summary>
  /// The 'Creator' abstract class
  /// </summary>
  abstract class Creator
  {
    public abstract Product FactoryMethod();
  }
```

```

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class ConcreteCreatorA : Creator
  {
    public override Product FactoryMethod()
    {
      return new ConcreteProductA();
    }
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class ConcreteCreatorB : Creator
  {
    public override Product FactoryMethod()
    {
      return new ConcreteProductB();
    }
  }
}
```

Output

```
Created ConcreteProductA
Created ConcreteProductB
```

Real-world code in C#

This real-world code demonstrates the Factory method offering flexibility in creating different documents. The derived Document classes Report and Resume instantiate extended versions of the Document class. Here, the Factory Method is called in the constructor of the Document base class.

```


using System;
using System.Collections.Generic;
```

```

namespace DoFactory.GangOfFour.Factory.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Factory Method Design Pattern.
  /// </summary>
  class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    static void Main()
    {
      // Note: constructors call Factory Method
      Document[] documents = new Document[2];

      documents[0] = new Resume();
      documents[1] = new Report();

      // Display document pages
      foreach (Document document in documents)
      {
        Console.WriteLine("\n" + document.GetType().Name + "--");
        foreach (Page page in document.Pages)
        {
          Console.WriteLine(" " + page.GetType().Name);
        }
      }

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Product' abstract class
  /// </summary>
  abstract class Page
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class SkillsPage : Page
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class EducationPage : Page
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class ExperiencePage : Page
  {
  }

  /// <summary>
  /// A 'ConcreteProduct' class
  /// </summary>
  class IntroductionPage : Page
  {
  }
```

```

    /// <summary>
    /// A 'ConcreteProduct' class
    /// </summary>
    class ResultsPage : Page
    {
    }

    /// <summary>
    /// A 'ConcreteProduct' class
    /// </summary>
    class ConclusionPage : Page
    {
    }

    /// <summary>
    /// A 'ConcreteProduct' class
    /// </summary>
    class SummaryPage : Page
    {
    }

    /// <summary>
```

```
  /// A 'ConcreteProduct' class
  /// </summary>
  class BibliographyPage : Page
  {
  }

  /// <summary>
  /// The 'Creator' abstract class
  /// </summary>
  abstract class Document
  {
    private List<Page> _pages = new List<Page>();

    // Constructor calls abstract Factory method
    public Document()
    {
      this.CreatePages();
    }

    public List<Page> Pages
    {
      get { return _pages; }
    }
```

```csharp

    // Factory Method
    public abstract void CreatePages();
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class Resume : Document
  {
    // Factory Method implementation
    public override void CreatePages()
    {
      Pages.Add(new SkillsPage());
      Pages.Add(new EducationPage());
      Pages.Add(new ExperiencePage());
    }
  }

  /// <summary>
  /// A 'ConcreteCreator' class
  /// </summary>
  class Report : Document
```

```
 {
  // Factory Method implementation
  public override void CreatePages()
  {
   Pages.Add(new IntroductionPage());
   Pages.Add(new ResultsPage());
   Pages.Add(new ConclusionPage());
   Pages.Add(new SummaryPage());
   Pages.Add(new BibliographyPage());
  }
 }
}
```

Output

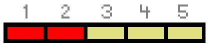
```
Resume -------
 SkillsPage
 EducationPage
 ExperiencePage

Report -------
 IntroductionPage
 ResultsPage
 ConclusionPage
```

Builder

Definition

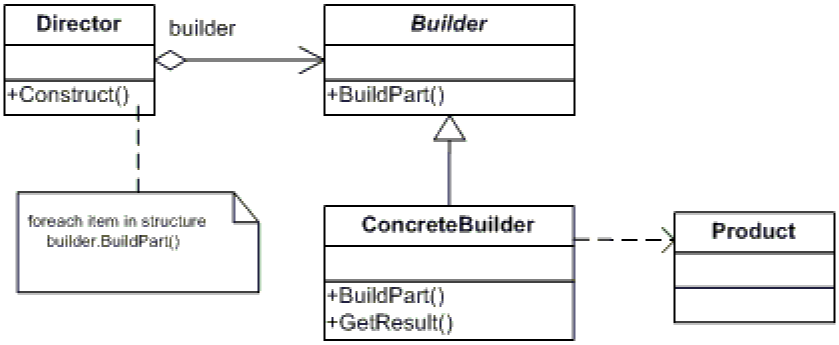
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Frequency of use:



Medium low

UML class diagram



Participants

The classes and objects participating in this pattern are:

- **Builder (VehicleBuilder)**
  - specifies an abstract interface for creating parts of a Product object
- **ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates

- provides an interface for retrieving the product

- **Director (Shop)**
  - constructs an object using the Builder interface
- **Product (Vehicle)**
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Structural code in C#

This structural code demonstrates the Builder pattern in which complex objects are created in a step-by-step fashion. The construction process can create different object representations and provides a high level of control over the assembly of the objects.

```


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Builder.Structural
```

```
{
  /// <summary>
  /// MainApp startup class for Structural
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
      // Create director and builders
      Director director = new Director();

      Builder b1 = new ConcreteBuilder1();
      Builder b2 = new ConcreteBuilder2();

      // Construct two products
      director.Construct(b1);
      Product p1 = b1.GetResult();
      p1.Show();

      director.Construct(b2);
      Product p2 = b2.GetResult();
      p2.Show();

      // Wait for user
      Console.ReadKey();
    }
  }

  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Director
  {
    // Builder uses a complex series of steps
    public void Construct(Builder builder)
    {
      builder.BuildPartA();
      builder.BuildPartB();
    }
  }

  /// <summary>
```

```
/// The 'Builder' abstract class
/// </summary>
abstract class Builder
{
  public abstract void BuildPartA();
  public abstract void BuildPartB();
  public abstract Product GetResult();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class ConcreteBuilder1 : Builder
{
  private Product _product = new Product();

  public override void BuildPartA()
  {
    _product.Add("PartA");
  }

  public override void BuildPartB()
  {
    _product.Add("PartB");
  }

  public override Product GetResult()
  {
    return _product;
  }
}

/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class ConcreteBuilder2 : Builder
{
  private Product _product = new Product();

  public override void BuildPartA()
  {
    _product.Add("PartX");
  }

  public override void BuildPartB()
  {
    _product.Add("PartY");
  }

  public override Product GetResult()
  {
    return _product;
  }
}

/// <summary>
/// The 'Product' class
/// </summary>
class Product
{
  private List<string> _parts = new List<string>();

  public void Add(string part)
  {
    _parts.Add(part);
  }

  public void Show()
  {
```

```
              Console.WriteLine("\nProduct Parts -------");
              foreach (string part in _parts)
                Console.WriteLine(part);
            }
          }
        }



```

Output

```
Product Parts -------
PartA
PartB

Product Parts -------
PartX
PartY
```

Real-world code in C#

This real-world code demonstates the Builder pattern in which different vehicles are assembled in a step-by-step fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

```


using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Builder.RealWorld
{
  /// <summary>
  /// MainApp startup class for Real-World
  /// Builder Design Pattern.
  /// </summary>
  public class MainApp
  {
    /// <summary>
    /// Entry point into console application.
    /// </summary>
    public static void Main()
    {
            VehicleBuilder builder;

            // Create shop with vehicle builders
            Shop shop = new Shop();

            // Construct and display vehicles
            builder = new ScooterBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            builder = new CarBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            builder = new MotorCycleBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            // Wait for user
            Console.ReadKey();
        }
    }

```

```
  /// <summary>
  /// The 'Director' class
  /// </summary>
  class Shop
  {
    // Builder uses a complex series of steps
    public void Construct(VehicleBuilder vehicleBuilder)
    {
      vehicleBuilder.BuildFrame();
      vehicleBuilder.BuildEngine();
      vehicleBuilder.BuildWheels();
      vehicleBuilder.BuildDoors();
    }
  }

  /// <summary>
  /// The 'Builder' abstract class
  /// </summary>
  abstract class VehicleBuilder
  {
    protected Vehicle vehicle;

    // Gets vehicle instance
    public Vehicle Vehicle
    {
      get { return vehicle; }
    }

    // Abstract build methods
    public abstract void BuildFrame();
    public abstract void BuildEngine();
    public abstract void BuildWheels();
    public abstract void BuildDoors();
  }

  /// <summary>
  /// The 'ConcreteBuilder1' class
  /// </summary>
  class MotorCycleBuilder : VehicleBuilder
  {
    public MotorCycleBuilder()
    {
      vehicle = new Vehicle("MotorCycle");
    }

    public override void BuildFrame()
```

```
      {
        vehicle["frame"] = "MotorCycle Frame";
      }

      public override void BuildEngine()
      {
        vehicle["engine"] = "500 cc";
      }

      public override void BuildWheels()
      {
        vehicle["wheels"] = "2";
      }

      public override void BuildDoors()
      {
        vehicle["doors"] = "0";
      }
    }


    /// <summary>
    /// The 'ConcreteBuilder2' class
    /// </summary>
    class CarBuilder : VehicleBuilder
    {
      public CarBuilder()
      {
        vehicle = new Vehicle("Car");
      }

      public override void BuildFrame()
      {
        vehicle["frame"] = "Car Frame";
      }

      public override void BuildEngine()
      {
        vehicle["engine"] = "2500 cc";
      }

      public override void BuildWheels()
      {
        vehicle["wheels"] = "4";
      }

      public override void BuildDoors()
      {
        vehicle["doors"] = "4";
      }
    }

    /// <summary>
    /// The 'ConcreteBuilder3' class
    /// </summary>
    class ScooterBuilder : VehicleBuilder
    {
      public ScooterBuilder()
      {
        vehicle = new Vehicle("Scooter");
      }

      public override void BuildFrame()
      {
        vehicle["frame"] = "Scooter Frame";
      }

      public override void BuildEngine()
      {
```

```
    vehicle["engine"] = "50 cc";
   }

   public override void BuildWheels()
   {
    vehicle["wheels"] = "2";
   }

   public override void BuildDoors()
   {
    vehicle["doors"] = "0";
   }
  }

  /// <summary>
  /// The 'Product' class
  /// </summary>
  class Vehicle
  {
   private string _vehicleType;
   private Dictionary<string,string> _parts =
    new Dictionary<string,string>();

   // Constructor
   public Vehicle(string vehicleType)
   {
    this._vehicleType = vehicleType;
   }

   // Indexer
   public string this[string key]
   {
    get { return _parts[key]; }
    set { _parts[key] = value; }
   }

   public void Show()
   {
    Console.WriteLine("\n---------------------------");
    Console.WriteLine("Vehicle Type: {0}", _vehicleType);
    Console.WriteLine(" Frame : {0}", _parts["frame"]);
    Console.WriteLine(" Engine : {0}", _parts["engine"]);
    Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
    Console.WriteLine(" #Doors : {0}", _parts["doors"]);
   }
  }
```

```
    }
```

Output

```
---------------------------
Vehicle Type: Scooter
 Frame  : Scooter Frame
 Engine : none
 #Wheels: 2
 #Doors : 0

---------------------------
Vehicle Type: Car
 Frame  : Car Frame
 Engine : 2500 cc
 #Wheels: 4
 #Doors : 4

---------------------------
Vehicle Type: MotorCycle
 Frame  : MotorCycle Frame
 Engine : 500 cc
 #Wheels: 2
 #Doors : 0
```