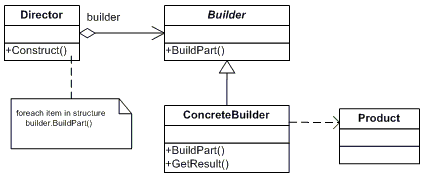
**Builder Design Pattern**

|  |  |
| --- | --- |
| http://www.dofactory.com/Images/redarrow.gif [definition](http://www.dofactory.com/Patterns/PatternBuilder.aspx#intent) http://www.dofactory.com/Images/redarrow.gif [UML diagram](http://www.dofactory.com/Patterns/PatternBuilder.aspx#UML) http://www.dofactory.com/Images/redarrow.gif [participants](http://www.dofactory.com/Patterns/PatternBuilder.aspx#participants) | http://www.dofactory.com/Images/redarrow.gif [sample code in C#](http://www.dofactory.com/Patterns/PatternBuilder.aspx#csharp) |

|  |  |  |  |
| --- | --- | --- | --- |
| definition  |  |  | | --- | --- | | http://www.dofactory.com/Images/pixel.gif | Separate the construction of a complex object from its representation so that the same construction process can create different representations.   Frequency of use: http://www.dofactory.com/Images/use_medium_low.gif  medium low | | http://www.dofactory.com/Images/552092_4.jpg |

[http://www.dofactory.com/Images/up.gifreturn to top](http://www.dofactory.com/Patterns/PatternBuilder.aspx)

### UML class diagram



[http://www.dofactory.com/Images/up.gifreturn to top](http://www.dofactory.com/Patterns/PatternBuilder.aspx)

### participants

    The classes and/or 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

[http://www.dofactory.com/Images/up.gifreturn to top](http://www.dofactory.com/Patterns/PatternBuilder.aspx)

### sample 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.

[**Hide code**](http://www.dofactory.com/Patterns/PatternBuilder.aspx#_self1)

|  |
| --- |
| // Builder pattern -- Structural example |
| 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 |

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.

[**Hide code**](http://www.dofactory.com/Patterns/PatternBuilder.aspx#_self2)

|  |
| --- |
| // Builder pattern -- Real World example |
| 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 |

This .NET optimized code demonstrates the same real-world situation as above but uses modern, built-in .NET features, such as, generics, reflection, object initializers, automatic properties, etc.