Articles » Development Lifecycle » Design and Architecture » Patterns and Practices

Design Patterns 1 of 3 - Creational Design Patterns

By **Kanasz Robert**, 28 Aug 2012

4.90 (173 votes)

Prize winner in Competition "Best C# article of July 2012"

Prize winner in Competition "Best overall article of July 2012"

Download Creational_StructuralExamples-noexe.zip - 24.9 KB

Download Creational_StructuralExamples.zip - 73.2 KB

Download Creational_RealWorldExamples-noexe.zip - 35.9 KB

Download Creational_RealWorldExamples.zip - 100.9 KB

Introduction

In software engineering, a design patterns are general reusable solutions to a software design problems you find again and again in a real life software development. Patterns are not a finished design that can be transformed directly into your code. They are only formal description or template how to solve specific problem. Patterns are best practices that you must implement yourself. Design patterns only describe interactions between classes and objects rather than large scale problems of overall software architecture.

Design patterns are very powerful tool for software developers, but you must keep in mind, that they shouldn’t be seen as prescriptive specifications for software. It is very important to understand the concepts of each pattern, rather than memorizing their concrete implementation, classes, methods and properties.

Every software developer should know how to apply this patterns appropriately. Using an inappropriate design pattern could cause bad performance of application or could increase complexity and maintainability of your code base.

In general software design patterns we can divide into three categories: creational patterns, structural patterns and behavioral patterns.

In this article I will talk about creational design patterns.

The Second[^] part of this series is about strucutral design patterns.

What are creational desing patterns?

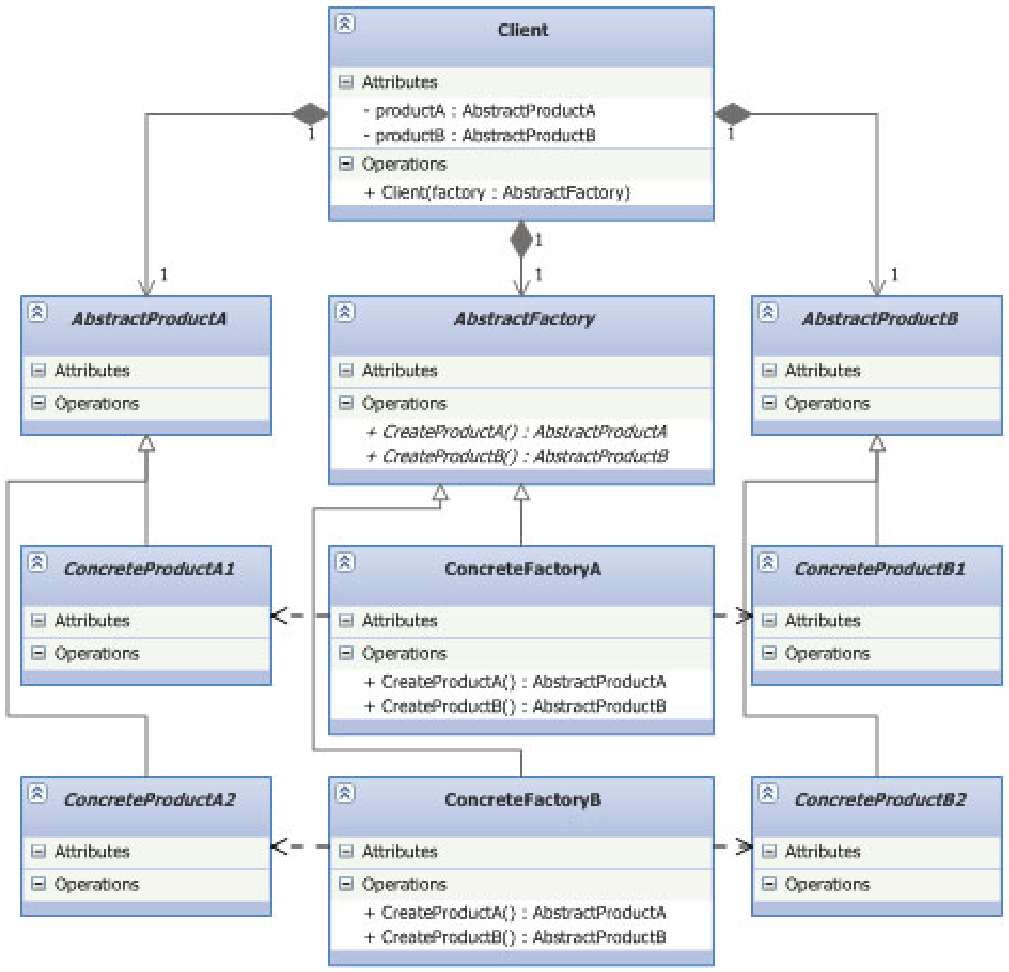
In software engineering, creational patterns are design patterns that deal with object creation mechanisms and are used in situations when basic form of object creation could result in design problems or increase complexity of a code base.

Abstract factory

The abstract factory pattern is a design pattern that allows for the creation of groups of related objects without the requirement of specifying the exact concrete classes that will be used. One of a number of factory classes generates the object sets.

The abstract factory pattern is one of the Gang of Four design patterns which belongs to the creational patterns group. Factories create different types of concrete objects. A Factory now represents a "family" of objects that it can create. This family of objects created by factory is determined at run-time according to the selection of concrete factory class. In this case client doesn’t know which concrete object it gets from each of factories, since it used only the generic interfaces of their products. This pattern is very good when you need to separate the details of objects instantiation. In general factories may have more than one factory method. Each factory method encapsulates the new operator and the concrete, platform-specific, product classes. Each platform is then modeled with a factory derived class.

Structural code example



The UML diagram below describes an implementation of the abstract factory design pattern. This diagram consists of five classes:

- **Client:** this class uses interfaces declared by `AbstractFactory` and `AbstractProduct` classes.
- **AbstractFactory:** this is a abstract base for all concrete factory classes that will generate new related objects. For each type of objects that will be instantiated a method is included within this
- **ConcreteFactory:** this class inherits from `AbstractFactory` class. `ConcreteFactory` overrides methods of `AbstractFactory` that will generate a new related objects. In case when `AbstractFactroy` is an interface, this class must implement all members of factory interface.
- **AbstractProduct:** this class is base class for all types of objects that factory can create.
- **ConcreteProduct:** this is an concrete implementation of `AbstractProduct` class. There can by a multiple classes which derives from `AbstractProduct` class with specific functionality.

```
using System;

namespace AbstractFactory
{
    static class Program
    {

        static void Main()
        {
            AbstractFactory factory1 = new ConcreteFactory1();
            var client1 = new Client(factory1);
            client1.Run();
            AbstractFactory factory2 = new ConcreteFactory2();
            var client2 = new Client(factory2);
            client2.Run();
        }
    }

    class Client
    {
        private readonly AbstractProductA _abstractProductA;
        private readonly AbstractProductB _abstractProductB;

        public Client(AbstractFactory factory)
        {
            _abstractProductB = factory.CreateProductB();
            _abstractProductA = factory.CreateProductA();
        }

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


    abstract class AbstractProductA
    {
    }

    abstract class AbstractProductB
    {
        public abstract void Interact(AbstractProductA a);
    }

    class ProductA1 : AbstractProductA
    {
    }

    class ProductB1 : AbstractProductB
    {
        public override void Interact(AbstractProductA a)
        {
            Console.WriteLine(GetType().Name +
              " interacts with " + a.GetType().Name);
        }
    }

    class ProductA2 : AbstractProductA
    {
    }

    class ProductB2 : AbstractProductB
    {
        public override void Interact(AbstractProductA a)
        {
            Console.WriteLine(GetType().Name +
              " interacts with " + a.GetType().Name);
        }
    }

    abstract class AbstractFactory
    {
        public abstract AbstractProductA CreateProductA();
        public abstract AbstractProductB CreateProductB();
    }

    class ConcreteFactory1 : AbstractFactory
    {
        public override AbstractProductA CreateProductA()
        {
            return new ProductA1();
        }
        public override AbstractProductB CreateProductB()
        {
            return new ProductB1();
        }
    }

    class ConcreteFactory2 : AbstractFactory
    {
        public override AbstractProductA CreateProductA()
        {
            return new ProductA2();
        }
        public override AbstractProductB CreateProductB()
        {
            return new ProductB2();
        }
    }
}
```

Real world example

The following example demonstrate the creation of different types of cars of different manufacturers. I have created one abstract factory interface called `IVehicleFactory` (abstract factory) and two concrete factory implementations of it called `FordFactory` (concrete factory) and `MitsubishiFactory` (concrete factory). `IVehicleFactory` has three methods which returns `Vehicle` (abstract product) objects. `Vehicle` class is a base class for all concrete products.

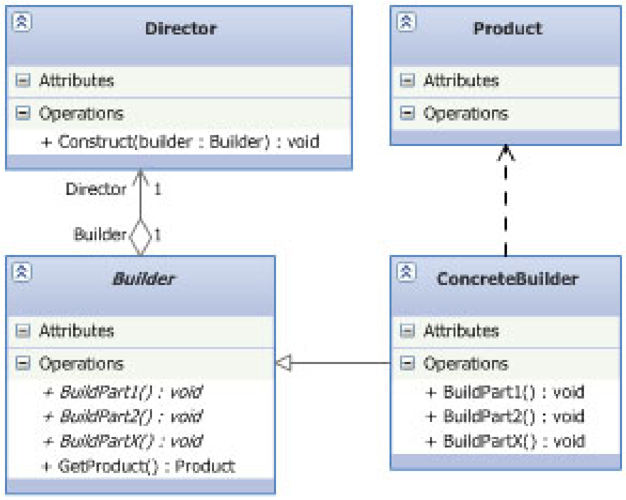
```
    public interface IVehicleFactory
    {
        Vehicle CreateEconomyCar();
        Vehicle CreateRacingCar();
        Vehicle CreateSUV();
    }

    public abstract class Vehicle
    {
        public string Model { get; set; }
        public string Engine { get; set; }
        public string Transmission { get; set; }
        public string Body { get; set; }
        public int Doors { get; set; }
        public List<string> Accessories = new List<string>();

        public abstract void ShowInfo();
    }

    class FordFactory:IVehicleFactory
    {
        public Vehicle CreateEconomyCar()
        {
            return new FordFocus();
        }

        public Vehicle CreateRacingCar()
        {
            return new FordGT1();
        }

        public Vehicle CreateSUV()
        {
            return new FordExplorer();
        }
    }

    public class MitsubishiFactory:IVehicleFactory
    {
        public Vehicle CreateEconomyCar()
        {
            return new MitsubishiI();
        }

        public Vehicle CreateRacingCar()
        {
            return new MitsubishiLancerEvoIX();
        }

        public Vehicle CreateSUV()
        {
            return new MitsubishiPajero();
        }
    }

    public class FordExplorer:Vehicle
    {
        public FordExplorer()
        {
            Model = "Ford Explorer";
            Engine = "4.0 L Cologne V6";
            Transmission = "5-speed M50D-R1 manual";
            Body = "SUV";
            Doors = 5;
            Accessories.Add("Car Cover");
            Accessories.Add("Sun Shade");
        }

        public override void ShowInfo()
        {
            Console.WriteLine("Model: {0}", Model);
            Console.WriteLine("Engine: {0}", Engine);
            Console.WriteLine("Body: {0}", Body);
            Console.WriteLine("Doors: {0}", Doors);
            Console.WriteLine("Transmission: {0}", Transmission);
            Console.WriteLine("Accessories:");
            foreach (var accessory in Accessories)
            {
                Console.WriteLine("\t{0}", accessory);
            }
        }
    }
    ...
```

Builder

The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions. The construction is controlled by a director object that only needs to know the type of object it is to create.

Builder pattern is another Gang of Four design pattern which belong to the creational design patterns family. The intent of the builder pattern is to separate the construction of a complex object from its representation. This pattern is used when complex object that need to be created is constructed by constituent parts that must by created in the same order or by using a specific algorithm.

Structural code example



The UML diagram below describes an implementation of the builder design pattern. This diagram consists of four classes:

- **Product:** represents the complex object that is being built.
- **Builder:** this is base class (or interface) for all builders and defines a steps that must be taken in order to correctly create an complex object (product). Generally each step is an abstract method that is overriden by concrete implementation.
- **ConcreteBuilder:** provides implementation for builder. Builder is an object able to create other complex objects (products).
- **Director:** represents class that controls algorithm used for creation of complex object.

```
static class Program
{
    static void Main()
    {
        Builder b1 = new ConcreteBuilder1();

        Director.Construct(b1);
        var p1 = b1.GetResult();
        p1.Show();
    }
}

static class Director
{
    public static void Construct(Builder builder)
```

```
        {
            builder.BuildPartA();
            builder.BuildPartB();
        }
    }

    abstract class Builder
    {
        public abstract void BuildPartA();
        public abstract void BuildPartB();
        public abstract Product GetResult();
    }

    class ConcreteBuilder1 : Builder
    {
        private readonly Product _product = new Product();

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

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

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

    class Product
    {
        private readonly List<string> _parts = new List<string>();

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

        public void Show()
        {
            Console.WriteLine("Parts:");
            foreach (string part in _parts)
                Console.WriteLine("\t"+part);
        }
    }
```

Real world example

Now let’s take a look at real world example. As a real world example I choose vehicle manufacturing. In this example I have created abstract class `VehicleBuilder` (Builder) which is a base class for all concrete builder classes (`FordExplorerBuilder` and `LincolnAviatorBuilder`). This class has method `CreateVehicle` which instantiates protected `_vehicle` field of type `Vehicle` (Product). Other methods are abstract and must be overriden by concrete builder implementation. This methods set properties of `_vehicle`. The last class `VehicleCreator` plays role as Director. It has constructor with one `VehicleBuilder` parameter and method `CreateVehicle` which controls steps for creation of object and filling it’s properties. Method `GetVehicle` returns fully constructed `Vehicle` object.

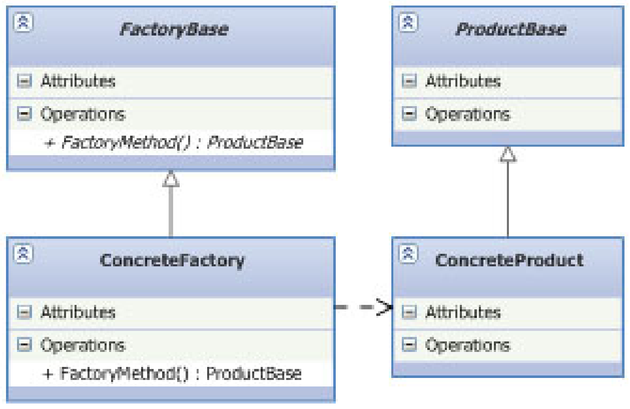
```
    class Program
    {
        static void Main()
        {
            var vehicleCreator = new VehicleCreator(new FordExplorerBuilder());
            vehicleCreator.CreateVehicle();
            var vehicle = vehicleCreator.GetVehicle();
            vehicle.ShowInfo();

            Console.WriteLine("---------------------------------------------");

            vehicleCreator = new VehicleCreator(new LincolnAviatorBuilder());
            vehicleCreator.CreateVehicle();
            vehicle = vehicleCreator.GetVehicle();
            vehicle.ShowInfo();

        }
    }

    public abstract class VehicleBuilder
    {
        protected Vehicle _vehicle;

        public Vehicle GetVehicle()
        {
            return _vehicle;
        }

        public void CreateVehicle()
        {
            _vehicle = new Vehicle();
        }

        public abstract void SetModel();
        public abstract void SetEngine();
        public abstract void SetTransmission();
        public abstract void SetBody();
        public abstract void SetDoors();
        public abstract void SetAccessories();
    }
    ...
    class FordExplorerBuilder : VehicleBuilder
    {
        public override void SetModel()
        {
            _vehicle.Model = "Ford Explorer";
        }

        public override void SetEngine()
        {
            _vehicle.Engine = "4.0 L Cologne V6";
        }

        public override void SetTransmission()
        {
            _vehicle.Transmission = "5-speed M5OD-R1 manual";
        }

        public override void SetBody()
        {
            _vehicle.Body = "SUV";
        }

        public override void SetDoors()
        {
            _vehicle.Doors = 5;
        }

        public override void SetAccessories()
        {
            _vehicle.Accessories.Add("Car Cover");
            _vehicle.Accessories.Add("Sun Shade");
        }
    }
    ...
    class LincolnAviatorBuilder : VehicleBuilder
    {
        public override void SetModel()
        {
            _vehicle.Model = "Lincoln Aviator";
        }

        public override void SetEngine()
        {
            _vehicle.Engine = "4.6 L DOHC Modular V8";
        }

        public override void SetTransmission()
        {
            _vehicle.Transmission = "5-speed automatic";
        }

        public override void SetBody()
        {
            _vehicle.Body = "SUV";
        }

        public override void SetDoors()
        {
            _vehicle.Doors = 4;
        }

        public override void SetAccessories()
        {
            _vehicle.Accessories.Add("Leather Look Seat Covers");
            _vehicle.Accessories.Add("Chequered Plate Racing Floor");
            _vehicle.Accessories.Add("4x 200 Watt Coaxial Speekers");
            _vehicle.Accessories.Add("500 Watt Bass Subwoofer");
        }
    }
    ...
    public class VehicleCreator
    {
        private readonly VehicleBuilder _builder;

        public VehicleCreator(VehicleBuilder builder)
        {
            _builder = builder;
        }

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

        public Vehicle GetVehicle()
        {
            return _builder.GetVehicle();
        }
    }
    ...
    public class Vehicle
    {
        public string Model { get; set; }
        public string Engine { get; set; }
        public string Transmission { get; set; }
        public string Body { get; set; }
        public int Doors { 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("Doors: {0}", Doors);
            Console.WriteLine("Transmission: {0}", Transmission);
            Console.WriteLine("Accessories:");
            foreach (var accessory in Accessories)
            {
                Console.WriteLine("\t{0}",accessory);
            }
        }
    }
```

Factory method

The factory method pattern is a design pattern that allows for the creation of objects without specifying the type of object that is to be created in code. A factory class contains a method that allows determination of the created type at run-time.

This is a third of Gang of Four patterns. It also belongs to the creational patterns group. This pattern is also known as Virtual Constructor pattern. The factory method pattern defines an interface for creating an object and leaves the choice of type to the subclasses. Factory method design pattern makes a design more customizable and only a little complicated. Other design pattern require new classes, whereas factory method only requires a new operation.

Structural code example



The UML diagram below describes an implementation of the factory method design pattern. This diagram consists of four classes:

- **FactoryBase**: this is an abstract class for the concrete factory classes which will return new objects. In some cases it could be a simple interface containing the signature for the factory method. This class contains `FactoryMethod` which returns a `ProductBase` object.
- **ConcreteFactory**: represents concrete implementation of factory. Usually this class overrides the generating `FactoryMethod` and returns a `ConcreteProduct` object.
- **ProductBase**: this is a base class for all products created by concrete factories. In some cases it could be a simple interface.
- **ConcreteProduct**: this is a concrete implementation of `ProducBase`. Concrete product classes can include specific functionality. This objects are created by factory methods.

```
static class Program
{
    static void Main()
    {
        FactoryBase factory = new ConcreteFactory();
        ProductBase product = factory.FactoryMethod(1);
        product.ShowInfo();
        product = factory.FactoryMethod(2);
        product.ShowInfo();
    }
}

public abstract class FactoryBase
{
    public abstract ProductBase FactoryMethod(int type);
}

public class ConcreteFactory : FactoryBase
{
    public override ProductBase FactoryMethod(int type)
    {
        switch (type)
        {
            case 1:
                return new ConcreteProduct1();
            case 2:
                return new ConcreteProduct2();
            default:
                throw new ArgumentException("Invalid type.", "type");
        }
    }
}
```

```
    public abstract class ProductBase
    {
        public abstract void ShowInfo();
    }

    public class ConcreteProduct1 : ProductBase {
        public override void ShowInfo()
        {
            Console.WriteLine("Product1");
        }
    }

    public class ConcreteProduct2 : ProductBase {
        public override void ShowInfo()
        {
            Console.WriteLine("Product2");
        }
    }
```

Real world example

In this example I have chosen the vehicle manufacturing example again. We have one interface `IVehicleFactory` with one method `CreateVehicle` which returns `Vehicle` object. Another two classes `FordExplorerFactory` and `LincolnAviatorFactory` are concrete implement this interface. In case of `FordExplorerFactory`, method `CreateVehicle` returns `FordExplorer` object which derives from abstract `Vehicle` class and overrides `ShowInfo` method. This method only displays information about vehicle. `FordExploredClass` has one default constructor which fills properties of this object.

In case of `LincolnAviatorFactory` is a situation a slightly bit different. Method `CreateVehicle` returns `LincolnAviator` object, but this object has one constructor with parameters which values are used to fill object’s properties.

This example demonstrates how to use different factories for creating a different types of objects.

```
    class Program
    {
        static void Main(string[] args)
        {
            IVehicleFactory factory =
GetFactory("FactoryMethodPattern.ConcreteFactories.LincolnAviatorFactory");

            var lincolnAviator = factory.CreateVehicle();

            lincolnAviator.ShowInfo();

            factory =
GetFactory("FactoryMethodPattern.ConcreteFactories.FordExplorerFactory");

            var fordExplorer = factory.CreateVehicle();

            fordExplorer.ShowInfo();
        }

        static IVehicleFactory GetFactory(string factoryName)
        {
            return Assembly.GetExecutingAssembly().CreateInstance(factoryName) as 
IVehicleFactory;
        }
    }

    public interface IVehicleFactory
    {
        Vehicle CreateVehicle();
    }

    public abstract class Vehicle
    {
        public string Model { get; set; }
        public string Engine { get; set; }
        public string Transmission { get; set; }
        public string Body { get; set; }
        public int Doors { get; set; }
        public List<string> Accessories = new List<string>();

        public abstract void ShowInfo();
    }

    public class FordExplorerFactory:IVehicleFactory
    {
        public Vehicle CreateVehicle()
        {
            return new FordExplorer();
        }
    }

    public class LincolnAviatorFactory:IVehicleFactory
    {
        public Vehicle CreateVehicle()
        {
            return new LincolnAviator("Lincoln Aviator",
                "4.6 L DOHC Modular V8",
                "5-speed automatic",
                "SUV",4);
        }
    }

    public class FordExplorer:Vehicle
    {
        public FordExplorer()
        {
            Model = "Ford Explorer";
            Engine = "4.0 L Cologne V6";
            Transmission = "5-speed M50D-R1 manual";
            Body = "SUV";
            Doors = 5;
            Accessories.Add("Car Cover");
            Accessories.Add("Sun Shade");
        }

        public override void ShowInfo()
        {
            Console.WriteLine("Model: {0}", Model);
            Console.WriteLine("Engine: {0}", Engine);
            Console.WriteLine("Body: {0}", Body);
            Console.WriteLine("Doors: {0}", Doors);
            Console.WriteLine("Transmission: {0}", Transmission);
            Console.WriteLine("Accessories:");
            foreach (var accessory in Accessories)
            {
                Console.WriteLine("\t{0}", accessory);
            }
        }
    }

    public class LincolnAviator:Vehicle
    {
        public LincolnAviator(string model, string engine, string transmission, string body, 
int doors)
        {
            Model = model;
            Engine = engine;
            Transmission = transmission;
            Body = body;
            Doors = doors;
            Accessories.Add("Leather Look Seat Covers");
            Accessories.Add("Chequered Plate Racing Floor");
            Accessories.Add("4x 200 Watt Coaxial Speekers");
            Accessories.Add("500 Watt Bass Subwoofer");
        }

        public override void ShowInfo()
        {
            Console.WriteLine("Model: {0}", Model);
            Console.WriteLine("Engine: {0}", Engine);
            Console.WriteLine("Body: {0}", Body);
            Console.WriteLine("Doors: {0}", Doors);
            Console.WriteLine("Transmission: {0}", Transmission);
            Console.WriteLine("Accessories:");
            foreach (var accessory in Accessories)
            {
                Console.WriteLine("\t{0}", accessory);
            }
        }
    }
```

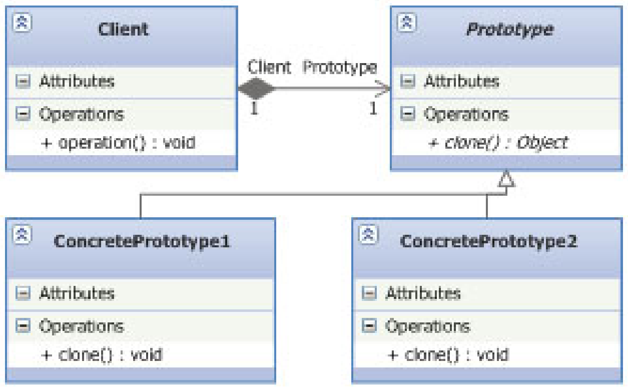
Prototype

The prototype design pattern is a design pattern that is used to instantiate a class by copying, or cloning, the properties of an existing object. The new object is an exact copy of the prototype but permits modification without altering the original.

This is another Gang of Four creational pattern. As programmer I'm absolutely sure that you found yourself create objects using new keyword many times and in some cases, when objects has a lot of properties which must be filled it was so boring and quantity of time you spent on this single task was enormous. I thing you agree when I say that today's programming is all about the costs and saving of resources (especially time) is a big issue. Sometimes you can find another way how to instantiate a new object using existing objects. This type of object creation is called cloning. This practise is very useful when the construction of a new object using new keyword is inefficient.

When original object is cloned, the new object is a shallow (or sometimes called deep) copy, This copy duplicates all of the properties and fields of original object. If a property is reference type, the reference is copied too.

Structural code example



The UML diagram below describes an implementation of the prototype design pattern. This diagram consits of two type of classes:

- **Prototype:** represents an abstract base class from the objects that can be cloned. This class contains single virtual method `Clone()` that returns prototype object. .NET framework includes interface named `ICloneable` which creates a new instance of a class with the same value as an existing instance.
- **ConcretePrototype:** this class inherits from `Prototype` base class and includes additional functionality. In this class is also overriden `Clone()` method.

```
    static class Program
    {
        static void Main()
        {
            ConcretePrototype prototype = new ConcretePrototype("1");
            ConcretePrototype clone = (ConcretePrototype)prototype.Clone();
            Console.WriteLine("Cloned: {0}", clone.Id);
        }
    }

    abstract class Prototype
    {
        private readonly string _id;

        protected Prototype(string id)
        {
            _id = id;
        }

        public string Id
        {
            get { return _id; }
        }

        public abstract Prototype Clone();
    }

    class ConcretePrototype : Prototype
    {
        public ConcretePrototype(string id)
            : base(id)
        {
        }
        public override Prototype Clone()
        {
            return (Prototype)MemberwiseClone();
        }
    }
```

Real world example

In this example I have created two concrete prototype classes: Commander and Infantry. Both of this classes inherites from StormTrooper class. StormTrooper class has two public methods: FirePower and Armor and one public method Clone which returns a shallow copy of object.

```
    class Program
    {
        static void Main()
        {
            var stormCommander = new Commander();
            var infantry = new Infantry();


            var stormCommander2 = stormCommander.Clone() as Commander;
            var infantry2 = infantry.Clone() as Infantry;


            if (stormCommander2 != null)
                Console.WriteLine("Firepower: {0}, Armor:
{1}",stormCommander2.FirePower,stormCommander2.Armor);

            if (infantry2 != null)
                Console.WriteLine("Firepower: {0}, Armor: {1}", infantry2.FirePower,
infantry2.Armor);
        }
    }

    public abstract class StormTrooper:ICloneable
    {
        public int FirePower { get; set; }
        public int Armor { get; set; }

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

    public class Commander:StormTrooper
    {
        public Commander()
        {
            Armor = 15;
            FirePower = 20;
        }
    }

    public class Infantry : StormTrooper
    {
        public Infantry()
        {
            FirePower = 10;
            Armor = 9;
        }
    }
```

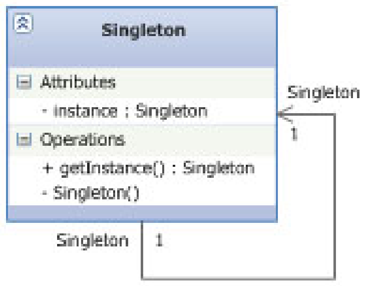
Singleton

The singleton pattern is a design pattern that is used to ensure that a class can only have one concurrent instance. Whenever additional objects of a singleton class are required, the previously created, single instance is provided.

In some cases it is a very important to have only one instance for a concrete class. Especially when you need only a single global point of access to a limited resource and when creating a global variable which will be copied could lead to multiple access points and caused some date inconsistencies. In ASP.NET `HttpContext` class is a nice example of singleton.

Base idea of singleton pattern is to centralize management of internal or external resources and to provide a global point of access to this resources.

Structural code example



The UML diagram below describes an implementation of the singleton design pattern. This diagram consists of only one class:

- **Singleton:** in this class is a static method called `GetSingleton` which returns the single instance held in private variable.

```
    static class Program
    {
        static void Main()
        {
            var singleton1 = Singleton.GetSingleton();
            var singleton2 = Singleton.GetSingleton();

            Console.WriteLine(singleton1==singleton2);

        }
    }

    public sealed class Singleton
    {
        private static volatile Singleton _instance;
        private static readonly object _lockThis = new object();

        private Singleton() { }

        public static Singleton GetSingleton()
        {
            if (_instance == null)
            {
                lock (_lockThis)
                {
                    if (_instance == null) _instance = new Singleton();
                }
            }
            return _instance;
        }
    }
```

Real world example

In this example I have created one singleton class called ApplicationState. This class has two public properties: `LoginId` and `MaxSize` which represents a current state of application. To ensure that this state is only held in a single instance, the class uses singleton design pattern.

When you create an multi thread application where you need some singletons, you must ensure, that your singletons are thread safe. This could be done by lock clause in `GetSingleton` method on a place, where you return an object.

```
    class Program
    {
        static void Main()
        {
            var state = ApplicationState.GetState();
            state.LoginId = "kanasz";
            state.MaxSize = 1024;

            var state2 = ApplicationState.GetState();
            Console.WriteLine(state2.LoginId);
            Console.WriteLine(state2.MaxSize);
            Console.WriteLine(state == state2);
        }
    }

    public sealed  class ApplicationState
    {
        private static volatile ApplicationState _instance;
        private static object _lockThis = new object();

        private ApplicationState() { }

        public static ApplicationState GetState()
        {
            if (_instance == null)
            {
                lock (_lockThis)
                {
                    if (_instance == null) _instance = new ApplicationState();
                }
            }
            return _instance;
        }

        // State Information
        public string LoginId { get; set; }
        public int MaxSize { get; set; }
    }
```

Additional Resources

Here is the list of additional resources I advice to check:

- http://www.blackwasp.co.uk/Default.aspx
- http://www.dofactory.com/Patterns/Patterns.aspx
- http://www.go4expert.com/forums/showthread.php?t=5127

History

- 30 July - Original version posted
- 8 August - Singleton pattern updated - double check implemented

License

About the Author

Kanasz Robert
Architect The Staffing Edge &
Marwin Cassovia Soft
Slovakia

My name is Robert Kanasz and I have been working with ASP.NET, WinForms and C# for several years.

MCTS - .NET Framework 3.5, ASP.NET Applications
- SQL Server 2008, Database Development
- SQL Server 2008, Implementation and Maintenance
- .NET Framework 4, Data Access
- .NET Framework 4, Service Communication Applications
- .NET Framework 4, Web Applications
MCPD - ASP.NET Developer 3.5
- Web Developer 4
MCITP - Database Administrator 2008
- Database Developer 2008

Open source projects: DBScripter - Library for scripting SQL Server database objects

Please, do not forget vote

Comments and Discussions

137 messages have been posted for this article Visit **http://www.codeproject.com/Articles/430590/Design-Patterns-1-of-3-Creational-Design-Patterns** to post and view comments on this article, or click **here** to get a print view with messages.