Design Patterns In C# .NET (2023)

Design patterns provide general solutions or a flexible way to solve common design problems. This article introduces design patterns and how design patterns are implemented in C# and .NET.

Before starting with design patterns in .NET, let's understand the meaning of design patterns and why they are useful in software architecture and programming.

**What are Design Patterns in Software Development?**

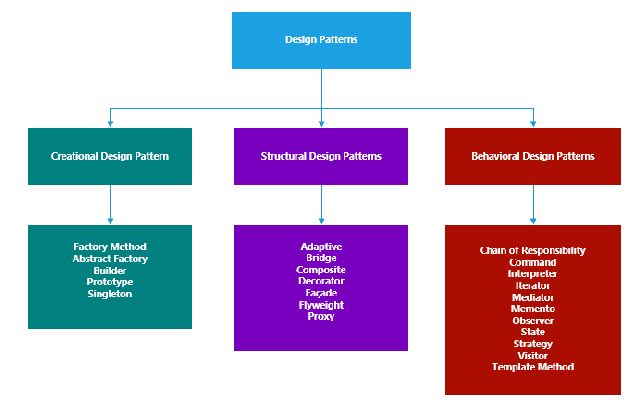
Design Patterns in the object-oriented world are a reusable solution to common software design problems that repeatedly occur in real-world application development. It is a template or description of how to solve problems that can be used in many situations.

"*A pattern is a recurring solution to a problem in a context.*"

"*Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice.*" - Christopher Alexander, *A Pattern Language*.

Developers use patterns for their specific designs to solve their problems. Pattern choice and usage among various design patterns depend on individual needs and concerns. Design patterns are a very powerful tool for software developers. It is important to understand design patterns rather than memorizing their classes, methods, and properties. Learning how to apply patterns to specific problems is also important to get the desired result. This will require continuous practice using and applying design patterns in software development. First, identify the software design problem, then see how to address these problems using design patterns and determine the best-suited design problem to solve the problem.

There are 23 design patterns, also known as Gang of Four (GoF) design patterns. The Gang of Four is the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". These 23 patterns are grouped into three main categories:



Creational Design Pattern

1. Factory Method
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton

Structural Design Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

Behavioral Design Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Visitor
11. Template Method

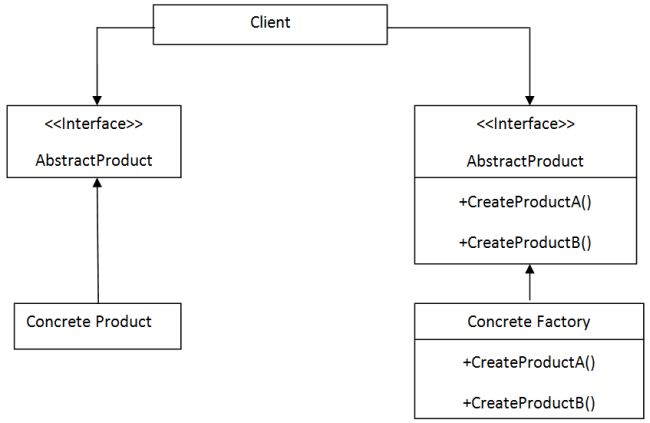
In this article, we learn and understand Creational Design Patterns in detail, including a UML diagram, template source code, and a real-world example in C#. Creational Design Patterns provide ways to instantiate a single object or group of related objects. These patterns deal with the object creation process in such a way that they are separated from their implementing system. That provides more flexibility in deciding which object needs to be created or instantiated for a given scenario. There are the following five such patterns.

**Abstract Factory**

This creates a set of related objects or dependent objects. The "family" of objects created by the factory is determined at run-time depending on the selection of concrete factory classes.

An abstract factory pattern acts as a super-factory that creates other factories. An abstract factory interface creates a set of related or dependent objects without specifying their concrete classes.

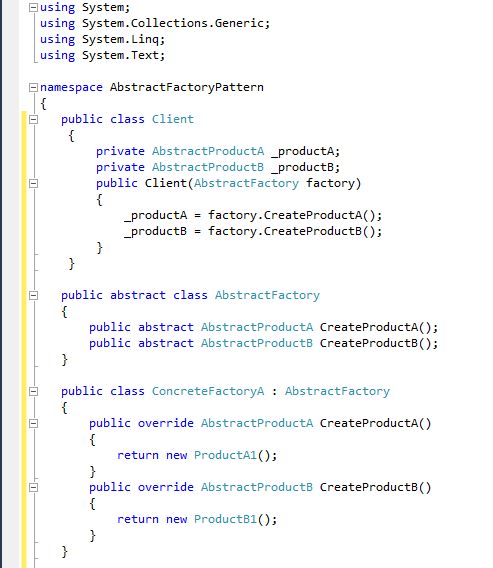
The UML class diagram below describes an implementation of the abstract factory design pattern.

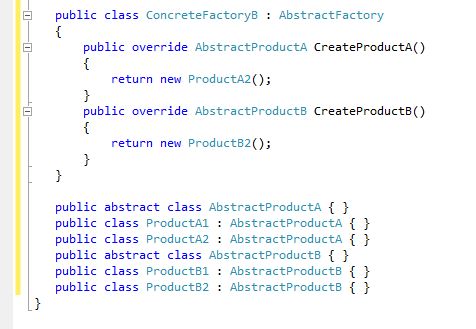


The classes, objects, and interfaces used in the above UML diagram are described below.

1. *Client* This class uses the Abstract Factory and Abstract Product interfaces to create a family of related objects.
2. *Abstract Factory* This is an interface that creates abstract products.
3. *Abstract Product* This is an interface that declares a type of product.
4. *Concrete factory*This class implements the abstract factory interface to create concrete products.
5. *Concrete Product* This class implements the abstract product interface to create products.

The following code shows the basic template code of the abstract factory design pattern implemented using C#:





In the above abstract factory design pattern, the source code template client has two private fields that hold the instances of abstract product classes. These objects will be accessed by inheriting their base class interface. When the client is instantiated, a concrete factory object is passed to its constructor and populated private fields of the client with appropriate data or values.

The Abstract factory is a base class for concrete factory classes that generate or create a set of related objects. This base class contains the definition of a method for each type of object that will be instantiated. The base class is declared Abstract so that other concrete factory subclasses can inherit it.

The concrete factory classes are inherited from the Abstract factory class and override the base class method to generate a set of related objects required by the client. Depending on the software or application requirements, there can be a specified number of concrete factory classes.

Abstractproduct is a base class for the types of objects that the factory class can create. There should be one base type for every distinct type of product required by the client.

The concrete product classes are inherited from Abstractproduct class. Each class contains specific functionality. Objects of these classes are generated from the Abstractfactory to populate the client.

A real-world example of an Abstract factory design pattern using C#

For example, consider a system that does the packaging and delivery of items for a web-based store. The company delivers two types of products. The first is a standard product placed in a box and delivered through the post with a simple label. The second is a delicate item that requires shock-proof packaging and is delivered via a courier. In this situation, two types of objects are required: a packaging object and a delivery documentation object. We could use two factories to generate these related objects. One factory will create packaging and other delivery objects for standard parcels. The second will create packaging and delivery objects for delicate parcels. **Class Client**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace AbstractFactoryPatternExample

{

public class Client

{

private Packaging \_packaging;

private DeliveryDocument \_deliveryDocument;

public Client(PacknDelvFactory factory)

{

\_packaging = factory.CreatePackaging();

\_deliveryDocument = factory.CreateDeliveryDocument();

}

public Packaging ClientPackaging

{

get { return \_packaging; }

}

public DeliveryDocument ClientDocument

{

get { return \_deliveryDocument; }

}

}

public abstract class PacknDelvFactory

{

public abstract Packaging CreatePackaging();

public abstract DeliveryDocument CreateDeliveryDocument();

}

public class StandardFactory : PacknDelvFactory

{

public override Packaging CreatePackaging()

{

return new StandardPackaging();

}

public override DeliveryDocument CreateDeliveryDocument()

{

return new Postal();

}

}

public class DelicateFactory : PacknDelvFactory

{

public override Packaging CreatePackaging()

{

return new ShockProofPackaging();

}

public override DeliveryDocument CreateDeliveryDocument()

{

return new Courier();

}

}

public abstract class Packaging { }

public class StandardPackaging : Packaging { }

public class ShockProofPackaging : Packaging { }

public abstract class DeliveryDocument { }

public class Postal : DeliveryDocument { }

public class Courier : DeliveryDocument { }

}

C#

Copy

**AbstractFactory Patterns Form**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

namespace AbstractFactoryPatternExample

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

GetClientObject();

}

void GetClientObject()

{

PacknDelvFactory sf = new StandardFactory();

Client standard = new Client(sf);

label1.Text = standard.ClientPackaging.GetType().ToString();

label2.Text = standard.ClientDocument.GetType().ToString();

PacknDelvFactory df = new DelicateFactory();

Client delicate = new Client(df);

label3.Text = delicate.ClientPackaging.GetType().ToString();

label4.Text = delicate.ClientDocument.GetType().ToString();

}

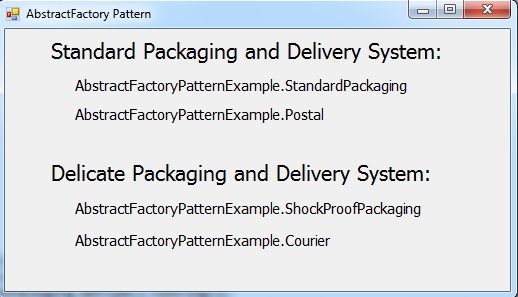
}

}

C#

Copy

**Output**



The example code above creates two client objects, each passing to a different type of factory constructor. Types of generated objects are accessed through the client's properties.

**Note** While studying abstract factory patterns, one question is, what are concrete classes? So I Googled that, and the following answers my question. A concrete class is nothing but a normal class that has all basic class features, like variables, methods, constructors, and so on. We can create an instance of the class in other classes.

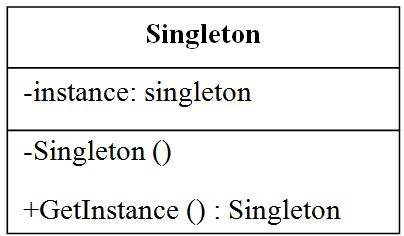
Here is a detailed article on [Abstract Factory Design Pattern In C#](https://www.c-sharpcorner.com/article/abstract-factory-design-pattern-in-c-sharp/)

Singleton Design Pattern

The Singleton design pattern is one of the simplest design patterns. This pattern ensures that the class has only one instance and provides a global point of access to it. The pattern ensures that only one object of a specific class is ever created. All further references to objects of the singleton class refer to the same underlying instance.

There are situations in a project where we want only one instance of the object to be created and shared among the clients. No client can create an instance from outside. It is more appropriate than creating a global variable since it may be copied and lead to multiple access points.

The UML class diagram below describes an implementation of the abstract factory design pattern:



The UML diagram above the "GetInstace" method should be declared static in the singleton patterns. This is because this method returns a single instance held in a private "instance" variable.  In the singleton pattern, all the methods and instances are defined as static. The static keyword ensures that only one instance of the object is created, and you can call methods of the class without creating an object.

The constructor of a class is marked as private. This prevents any external classes from creating new instances. The class is also sealed to prevent inheritance, which could lead to subclassing that breaks the singleton rules.

The following code shows the basic template code of the singleton design pattern implemented using C#.

**The eager initialization of singleton pattern**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace SingetonDesignPattern

{

public sealed class SingeltonTemp

{

private static SingeltonTemp \_instance;

private static object \_lockThis = new object();

private SingeltonTemp()

{

}

public static SingeltonTemp GetSingleton()

{

lock (\_lockThis)

{

if (\_instance == null) \_instance = new SingeltonTemp();

}

return \_instance;

}

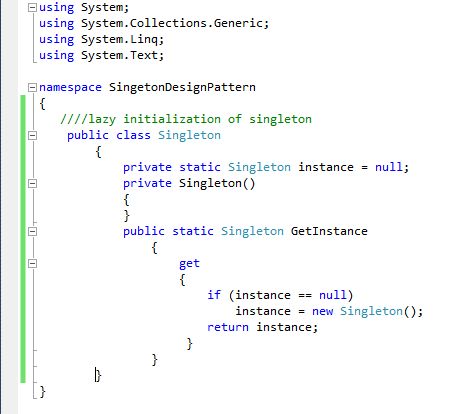
}

}

C#

Copy

**Lazy initialization of singleton pattern**

****

**Thread-safe (Double-checked Locking) initialization of singleton pattern**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace SingetonDesignPattern

{

public class Singleton

{

private static Singleton instance = null;

private Singleton()

{

}

private static object lockThis = new object();

public static Singleton GetInstance

{

get

{

lock (lockThis)

{

if (instance == null)

instance = new Singleton();

return instance;

}

}

}

}

}

C#

Copy

The code above shows the "lockThis" object and the use of locking within the "GetInstance" method. Since programs can be multithreaded, it is possible that two threads could request the singleton before the instance variable is initialized. By locking the dummy "lockThis" variable, all other threads will be blocked. This means that two threads will not be able to create their copies of the object simultaneously.

A real-world example of a Singleton design pattern using C#.net

I am trying to apply this pattern in my application where I want to maintain an application state for user login information and any other specific information required to be instantiated only once and held in only one instance.

**Class ApplicationState**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace SingetonDesignPattern

{

class ApplicationState

{

private static ApplicationState instance=null;

// State Information

public string LoginId

{ get; set;}

public string RoleId

{ get; set; }

private ApplicationState()

{

}

//Lock Object

private static object lockThis = new object();

public static ApplicationState GetState()

{

lock (lockThis)

{

if (ApplicationState.instance == null)

instance = new ApplicationState();

}

return instance;

}

}

}

C#

Copy

**Singleton pattern form**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

namespace SingetonDesignPattern

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

GetStateInfo();

}

void GetStateInfo()

{

ApplicationState state = ApplicationState.GetState();

state.LoginId="Kanchan";

state.RoleId= "Admin";

ApplicationState state2 = ApplicationState.GetState();

label3.Text = state2.LoginId;

label5.Text = state2.RoleId;

label6.Text = (state == state2).ToString();

}

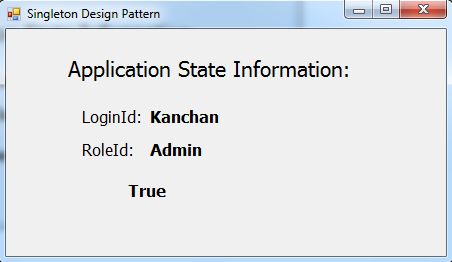
}

}

C#

Copy

**Output**

****

The preceding sample code creates two new variables and assigns the return value of the GetState method to each. They are then compared to check that they both contain the same values and a reference to the same object.