****

<https://www.youtube.com/watch?v=DfjdRitvyY4>

**PATRON FACTORY**

**ING. SISTEMAS DE INFORMACION Y GESTION**

**“ANALISIS Y DISEÑO ORIENTADO A OBJETOS”**

**INTEGRANTES:**

* FLORES PUMA, LUIS
* ZAVALA DIAZ, ANIBAL
* DE LA CRUZ, JOSELYN

**2017**

**SIMPLE FACTORY**

**Concepto:**

Una factoría es un objeto que crea otros objetos. El patrón de factoría simple es una clase Factoría en su estado más simple (en comparación con las dos clases de factoría: Abstracta y Método). En otras palabras, en un patrón de factoría simple, tenemos una clase factoría que tiene un método que devuelve diferentes tipos de objetos según una entrada dada.

Ejemplo de Código

/// <summary>

/// Simple Factory

/// Ejemplo con método de creación estático

/// </summary>

public class CreadorDePerros

{

   public static Mascota Create ()

   return new Perro();

   }

}

/// <summary>

/// Ejemplo de Simple Factory

/// </summary>

public class CreadorDeGatos

{

   public Mascota Create()

   {

   return new Gato();

   }

}

**Simple Factory y Factory Method**

Es fundamental recordar la intención de este patrón: Factory Method define una interfaz para crear objetos, pero deja que sean las subclases las que decidan qué clases instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

**Simple Factory y Abstract Factory**

Abstract Factory proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

**¿Qué interés práctico puede existir en crear una clase cuya función sea instanciar otras clases pudiendo dejarle el trabajo a la clase original?**

Esta forma de trabajar puede ser útil en algunos escenarios, pero el principal suele involucrar el **no saber qué objeto vamos a instanciar hasta el momento de la ejecución**. Valiéndonos del polimorfismo podremos utilizar una interfaz para alojar una referencia a un objeto que será instanciado por un tercero en lugar de dejar que sea el propio constructor del objeto el que proporcione la instancia.

|  |  |
| --- | --- |
| 1 | MotorDiesel motor = new MotorDiesel(); |

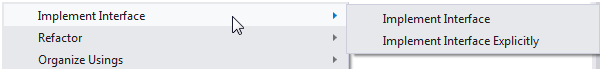
A esto otro:

|  |  |
| --- | --- |
| 1 | IMotor iMotor = MotorFactory.CreateInstance(tipoMotor); |

Por tanto, nuestro objetivo principal será la **encapsulación de la creación de objetos**. En este ejemplo de vehículos con un motor que está representado mediante una interfaz que será implementada por las clases MotorDiesel y MotorGasolina. La interfaz expondrá las siguientes propiedades y métodos:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | namespace Patterns.Factory.SimpleFactory.Interfaces  {      public interface IMotor      {          public int Estabilidad { get; set; }          public decimal ParMotor { get; set; }          public int Potencia { get; set; }          public decimal Rendimiento { get; set; }          public int VelocidadNominal { get; set; }            string ConsumirCombustible();          string InyectarCombustible();          string RealizarEscape();          string RealizarExpansion();      }  } |

A continuación codificaremos las dos clases que implementan la interfaz.



En primer lugar implementaremos el motor Diesel, cuyos métodos devolverán cadenas de texto con sus respectivos mensajes. Además implementará un método privado RealizarCombustion() para simbolizar el tratamiento del combustible.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class MotorDiesel : IMotor  {      #region IMotor Members        int Estabilidad { get; set; }      decimal ParMotor { get; set; }      int Potencia { get; set; }      decimal Rendimiento { get; set; }      int VelocidadNominal { get; set; }        public string ConsumirCombustible()      {          return RealizarCombustion();      }        public string InyectarCombustible(int cantidad)      {          return string.Format("MotorDiesel: Inyectados {0} ml. de Gasoil.", cantidad);      }        public string RealizarEscape()      {          return "MotorDiesel: Realizado escape de gases";      }        public string RealizarExpansion()      {          return "MotorDiesel: Realizada expansion";      }        #endregion        private string RealizarCombustion()      {          return "MotorDiesel: Realizada combustion del Gasoil";      }  } |

Realizamos lo propio con el motor de gasolina, salvando el hecho que, en lugar de realizar una combustión, realizará una explosión. Por lo tanto, el nombre del método privado será RealizarExplosion():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class MotorGasolina : IMotor  {      #region IMotor Members        public int Estabilidad { get; set; }      public decimal ParMotor { get; set; }      public int Potencia { get; set; }      public decimal Rendimiento { get; set; }      public int VelocidadNominal { get; set; }        public string ConsumirCombustible()      {          return RealizarExplosion();      }        public string InyectarCombustible(int cantidad)      {          return string.Format("MotorGasolina: Inyectados {0} ml. de Gasolina.", cantidad);      }        public string RealizarEscape()      {          return "MotorGasolina: Realizado escape de gases";      }        public string RealizarExpansion()      {          return "MotorGasolina: Realizada expansion";      }        #endregion        private string RealizarExplosion()      {          return "MotorGasolina: Realizada explosion de la Gasolina";      }  } |

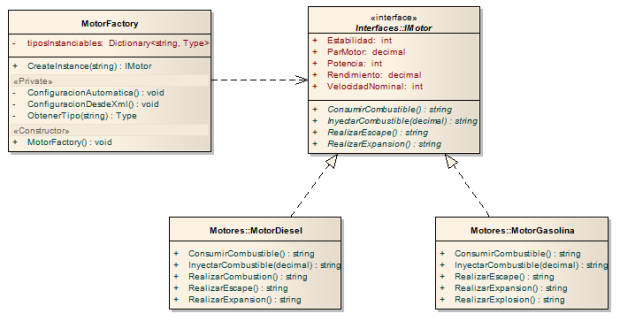
Finalmente, crearemos una clase MotorFactory que exhiba un método público CreateInstance(nombreClase) para instanciar, por ejemplo, un motor Diesel.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class MotorFactory  {      public IMotor CreateInstance(string tipoMotor)      {          IMotor resultado;            switch (tipoMotor)          {              case "MotorDiesel":                  resultado = new MotorDiesel() { Estabilidad = 100, Potencia = 40, Rendimiento = 800, VelocidadNominal = 0 };                  break;              default:                  resultado = null;                  break;          }            return resultado;      }  } |

Lo que estamos haciendo aquí no es otra cosa que delegar en MotorFactory la tarea de instanciar el motor. Ya tenemos el motor Diesel. Nos falta ahora el Gasolina. Para ello deberemos modificar la clase MotorFactory y añadir un Nuevo case a la sentencia switch:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class MotorFactory  {      public IMotor CreateInstance(string tipoMotor)      {          IMotor resultado;            switch (tipoMotor)          {              case "MotorDiesel":                  resultado = new MotorDiesel();                  break;              case "MotorGasolina":                  resultado = new MotorGasolina();                  break;              default:                  resultado = null;                  break;          }            return resultado;      }  } |

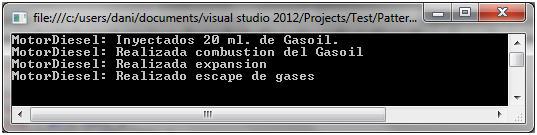
El diagrama de clases de esta primera factoría sería el siguiente:



Si quisiéramos obtener un motor diesel, por lo tanto, bastaría con el siguiente código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | static void Main(string[] args)  {      MotorFactory factory = new MotorFactory();        IMotor motorDiesel = factory.CreateInstance("MotorDiesel");        if (motorDiesel == null)          return;        Console.WriteLine(motorDiesel.InyectarCombustible(20));      Console.WriteLine(motorDiesel.ConsumirCombustible());      Console.WriteLine(motorDiesel.RealizarExpansion());      Console.WriteLine(motorDiesel.RealizarEscape());      Console.ReadLine();  } |

La ejecución daría origen al siguiente resultado:



**Método Factory**

Libera al desarrollador sobre la forma correcta de crear objetos. Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.

Muchas veces ocurre que una clase no puede anticipar el tipo de objetos que debe crear, ya que la jerarquía de clases que tiene requiere que deba delegar la responsabilidad a una subclase.

Características

• Clasificación: creacional

• Se lo conoce Virtual Constructor.

• Hace que un diseño sea más personalizable y sólo un poco más complicado. El método de fábrica sólo requiere una nueva operación.

• Es similar a la Abstract Factory, pero sin el énfasis en las familias.

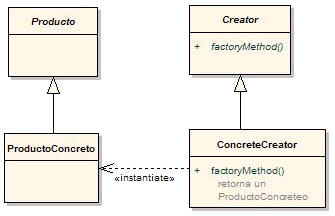
• Es de rutina, especificados por un marco arquitectónico, y luego implementar por parte del usuario la estructura.

• A este método se lo utiliza con el método plantilla.

Este patrón debe ser utilizado cuando:

* Una clase no puede anticipar el tipo de objeto que debe crear y quiere que sus subclases especifiquen dichos objetos.
* Hay clases que delegan responsabilidades en una o varias subclases. Una aplicación es grande y compleja y posee muchos patrones creacionales.

Diagrama UML.



Este patrón está compuesto por:

Creator: declara el método de fabricación (creación), que devuelve un objeto de tipo Product.

ConcretCreator: redefine el método de fabricación para devolver un producto.

ProductoConcreto: es el resultado final. El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

**Conclusiones:**

• El patrón Factory Method permite escribir aplicaciones que son más flexibles respecto de los tipos a utilizar difiriendo la creación de las instancias en el sistema a subclases que pueden ser extendidas a medida que evoluciona el sistema.

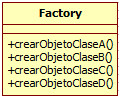
• Permite también encapsular el conocimiento referente a la creación de objetos. Factory Method hace también que el diseño sea más adaptable a cambio de sólo un poco más de complejidad.

• Inconveniente: Se obliga al cliente a definir subclases de la clase “Creador” sólo para crear unproducto concreto y esto puede no ser apropiado siempre

**Factory Abstracto**

Antes que nada debemos saber que existe un patrón llamado **Factory**el cual permite delegar en una clase la responsabilidad de crear objetos de otras clases, basados en esto podemos decir que el **Abstract Factory** es una nueva fase de **Factory**, teniendo así una Fábrica de Objetos donde la creación es transparente para la clase que los solicita.

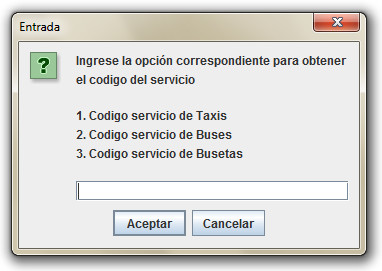
Debemos saber entonces que el patrón **Abstract Factory** nos brinda una interfaz para crear familias de Objetos relacionados sin tener que especificar sus clases concretas desde la clase cliente (Clase que los solicita).



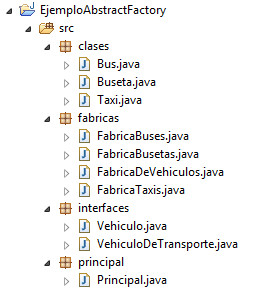
**El problema:** Se solicita la creación de diferentes vehículos de transporte (Buses, Busetas y Taxis) sin que se especifique en detalle la forma de su creación.

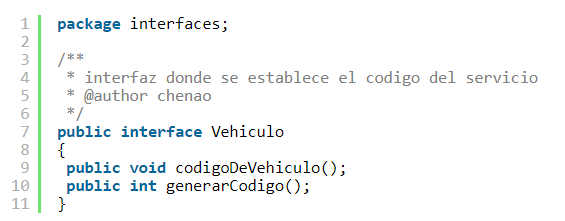
**La solución:** Utilizamos el patrón Abstract Factory para independizar la forma como crearemos los objetos, de esta manera creamos familias de objetos de tipo Vehículo delegando el proceso y sin tener que entrar en detalles desde la clase solicitante.

**La Aplicación:**

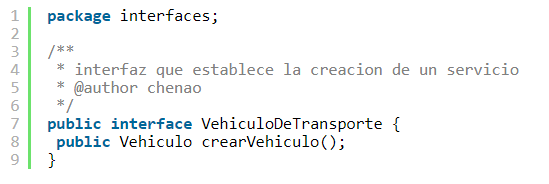


Veremos una aplicación pequeña que permite al usuario crear  familias de **vehículos**de 3 tipos diferentes, **Buses**, **Busetas**y **Taxis**, tan solo presentamos unas opciones de selección donde escogeremos que Objeto crear, he internamente  por medio del Patrón de Diseño ejecutamos el proceso de creación.

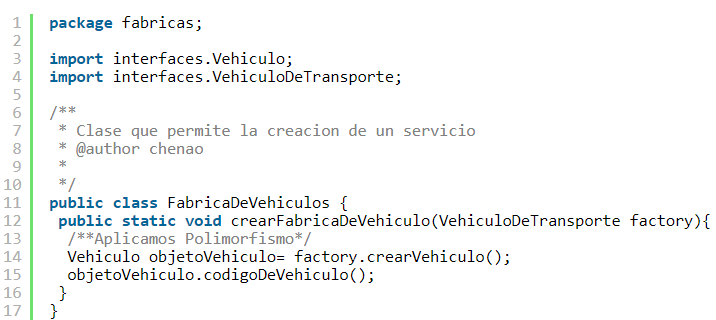


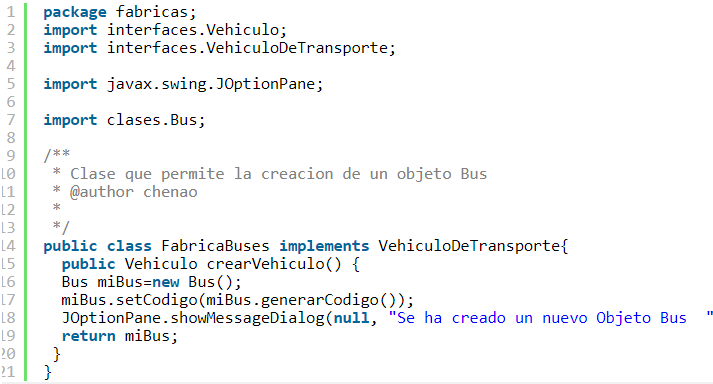
**Interface Vehículo.**

**Interface VehiculoDeTransporte.**

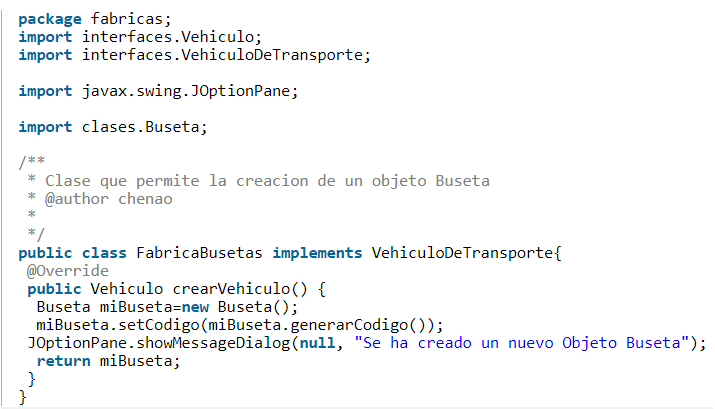


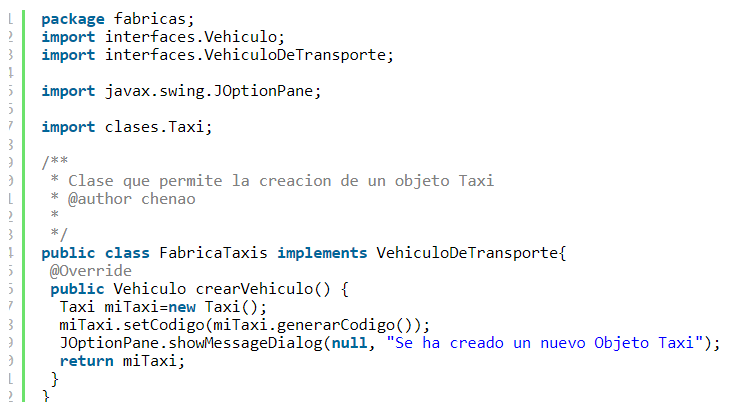
**Clase FabricaDeVehiculos.**



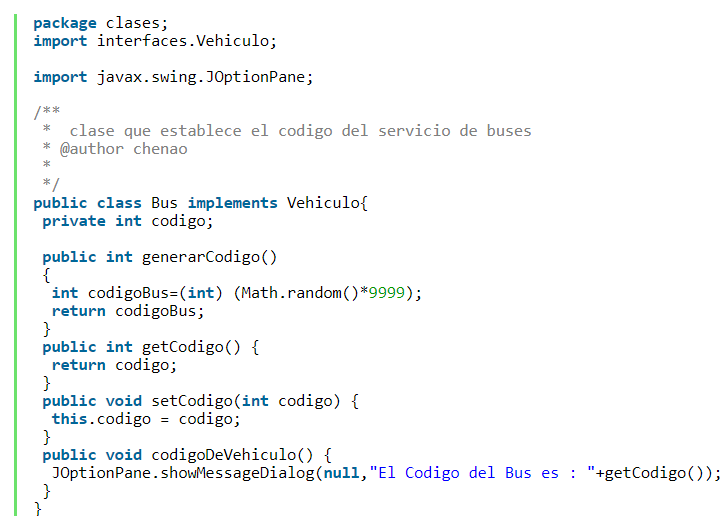
**Clase FabricaBuses.**

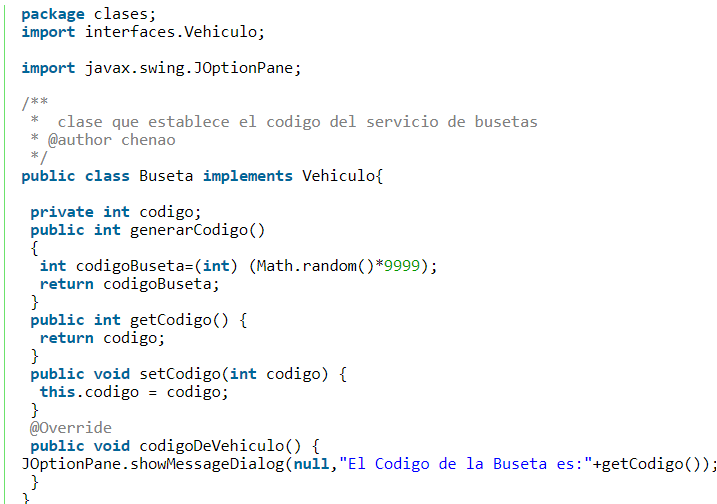
**Clase FabricaBusetas.**



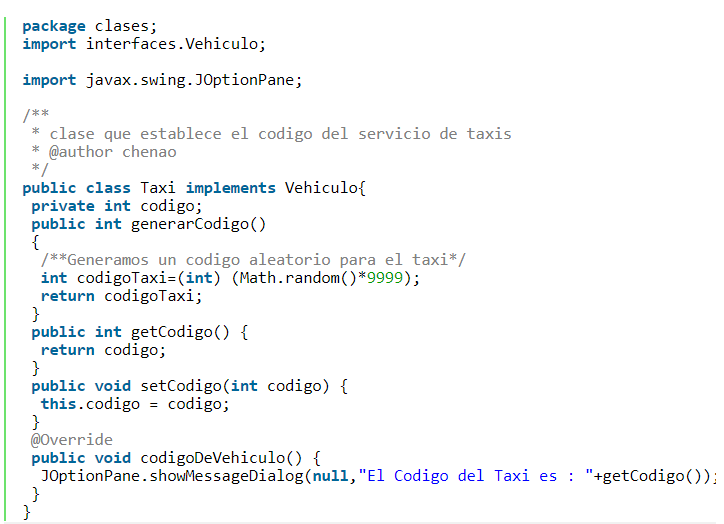
**Clase FabricaTaxis.**

**Clase Bus.**



**Clase Buseta.**

**Clase Taxi.**



**Clase Principal.**

