PATRONES DE DISEÑO

Factory Method:

Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán, es decir que nos provee de un método que las subclases se implementaran de la forma que necesite sin que nosotros necesitemos conocer como tenemos que hacer tal creación

Ejemplo 1:

Una aplicación de gestión logística para empresas de transporte marítimo

```
// La clase creadora declara el método fábrica que debe devolver
// un objeto de una clase de producto. Normalmente, las
// subclases de la creadora proporcionan la implementación de
// este método.
class Dialog is
    // La creadora también puede proporcionar cierta
    // implementación por defecto del método fábrica.
    abstract method createButton():Button

    // Observa que, a pesar de su nombre, la principal
    // responsabilidad de la creadora no es crear productos.
    // Normalmente contiene cierta lógica de negocio que depende
    // de los objetos de producto devueltos por el método
    // fábrica. Las subclases pueden cambiar indirectamente esa
    // lógica de negocio sobrescribiendo el método fábrica y
    // devolviendo desde él un tipo diferente de producto.
    method render() is
        // Invoca el método fábrica para crear un objeto de
        // producto.
        Button okButton = createButton()
        // Ahora utiliza el producto.
        okButton.onClick(closeDialog)
        okButton.render()


// Los creadores concretos sobrescriben el método fábrica para
// cambiar el tipo de producto resultante.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()


// La interfaz de producto declara las operaciones que todos los
// productos concretos deben implementar.
interface Button is
    method render()
```

```
    method onClick(f)

// Los productos concretos proporcionan varias implementaciones
// de la interfaz de producto.

class WindowsButton implements Button is
    method render(a, b) is
        // Representa un botón en estilo Windows.
    method onClick(f) is
        // Vincula un evento clic de OS nativo.

class HTMLButton implements Button is
    method render(a, b) is
        // Devuelve una representación HTML de un botón.
    method onClick(f) is
        // Vincula un evento clic de navegador web.

class Application is
    field dialog: Dialog

    // La aplicación elige un tipo de creador dependiendo de la
    // configuración actual o los ajustes del entorno.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // El código cliente funciona con una instancia de un
    // creador concreto, aunque a través de su interfaz base.
    // Siempre y cuando el cliente siga funcionando con el
    // creador a través de la interfaz base, puedes pasarle
    // cualquier subclase del creador.
    method main() is
        this.initialize()
        dialog.render()
```

Ejemplo 2:

Una aplicación para establecer de manera sencilla un color para un carro, donde el color por defecto es blanco, y leerlo

```
1. class Car {
2.   private $color = null;
3.   public function __construct() {
4.           $this->color = "white";
5.   }
6.   public function setColor($color) {
```

```
7.          $this->color = $color;
8.   }
9.   public function getColor() {
10.         return $this->color;
11.  }
12.}
```

Al aplicar el patrón factory method se presentan los métodos para los coches “rojos” (red) y “azules” (blue) así como un método privado para la propia creación de clase

```
1. class CarFactory {
2.   private function __construct() {
3.   }
4.   public static function getBlueCar() {
5.          return self::getCar("blue");
6.   }
7.   public static function getRedCar() {
8.          return self::getCar("red");
9.   }
10.  private static function getCar($color) {
11.         $car = new Car();
12.         $car->setColor($color);
13.         return $car;
14.  }
15.}
```

Singleton:

Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia. Es decir limita a uno el número de posibles instancias que podemos crear para una clase, proporcionando un acceso global a este objeto

Ejemplo 1:

la clase de conexión de la base de datos no tiene un constructor público, por lo que la única manera de obtener su objeto es invocando el método obtenerInstancia. Este método almacena en caché el primer objeto creado y lo devuelve en todas las llamadas siguientes.

```
// La clase Base de datos define el método `obtenerInstancia`
// que permite a los clientes acceder a la misma instancia de
// una conexión de la base de datos a través del programa.
class Database is
    // El campo para almacenar la instancia singleton debe
```

```
    // declararse estático.
    private static field instance: Database

    // El constructor del singleton siempre debe ser privado
    // para evitar llamadas de construcción directas con el
    // operador `new`.
    private constructor Database() is
        // Algún código de inicialización, como la propia
        // conexión al servidor de una base de datos.
        // ...

    // El método estático que controla el acceso a la instancia
    // singleton.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Garantiza que la instancia aún no se ha
                // inicializado por otro hilo mientras ésta ha
                // estado esperando el desbloqueo.
                if (Database.instance == null) then
                    Database.instance = new Database()
        return Database.instance

    // Por último, cualquier singleton debe definir cierta
    // lógica de negocio que pueda ejecutarse en su instancia.
    public method query(sql) is
        // Por ejemplo, todas las consultas a la base de datos
        // de una aplicación pasan por este método. Por lo
        // tanto, aquí puedes colocar lógica de regularización
        // (throttling) o de envío a la memoria caché.
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // La variable `bar` contendrá el mismo objeto que la
        // variable `foo`.
```

Ejemplo 2:

La clase llamada "SoyUnico" con un constructor que va a ser privado y un método llamado 'getSingletonInstance()' que será el encargado de crear una instancia de esta clase si no se ha creado todavía:

```
public class SoyUnico {
```

```
    private String nombre;
    private static SoyUnico soyUnico;

    // El constructor es privado, no permite que se genere un constructor por defecto
.
    private SoyUnico(String nombre) {
        this.nombre = nombre;
        System.out.println("Mi nombre es: " + this.nombre);
    }

    public static SoyUnico getSingletonInstance(String nombre) {
        if (soyUnico == null){
            soyUnico = new SoyUnico(nombre);
        }
        else{
            System.out.println("No se puede crear el objeto "+ nombre + " porque ya
existe un objeto de la clase SoyUnico");
        }

        return soyUnico;
    }

    // metodos getter y setter

}
```