

# PRACTICAS PATRONES DE DISEÑO

## PATRONES DE CREACIÓN:

- 1) **Patrón Singleton:** Diseñar una clase que permita internacionalizar una aplicación. Los ficheros de idioma está en la carpeta de prácticas. Utilizar un patrón Singleton.
- 2) **Patrón Factory Method:** A partir de una clase abstract "MedioTransporte" crear distintos medios de transporte según la opción elegida por el usuario. Coche, Avion, Barco.
- 3) **Patrón Abstract Factory:** Disponemos de **smartphones** y **tabletas**. De fabricantes: **Samsung, HTC, Nokia**.

```
Tablet phone from Samsung: Guru
Smart phone from Samsung: GalaxyS2
Tablet phone from HTC: Genie
Smart phone from HTC: Titan
Tablet phone from Nokia: Asha
Smart phone from Nokia: Lumia
```

- 4) **ABSTRACT FACTORY:** A partir del proyecto LABERINTO\_AF (incluye las clases del laberinto y las nuevas clases básicas. Hay que agregar las Factorías), se quiere poder crear distintos tipos de laberintos: Laberintos, LaberintosConBombas y LaberintosEncantados.

Implementar una clase FabricaLaberintos y dos subclases para los tipos ConBombas y Encantados. La fábrica de Laberintos dispondrá de los métodos:

- Laberinto \*hacerLaberinto(),
- Pared \*hacerPared(),
- Habitación \*hacerHabitacion(int n),
- Puerta \*hacerPuerta(Habitación \*h1, Habitación \*h2).

La idea es parametrizar el juego con una fábrica de laberintos que puede ser una de las 3 descritas.

En el caso de la fábrica de laberintos encantados, tendrá HabitacionEncantada y PuertaConHechizo (son subclases de Habitación y Puerta, respectivamente).

En el caso de la fábrica de laberintos con Bombas, tendrá:

HabitacionConBomba y ParedExplosionada (son subclases de Habitación y Pared, respectivamente).

En un main recoger los parámetros de la línea de comandos para crear el laberinto del tipo elegido (NORMAL, ENCANTADO o CON\_BOMBAS).

- 5) **PROTOTYPE:** A partir del proyecto patrón Abstract Factory. Se pretende desarrollar la clase FabricaPrototiposLaberinto que es subclase de FabricaLaberintos. Está fábrica se inicializa con unos objetos del laberinto (laberinto, pared, habitación y puerta) por defecto. También debemos implementar los métodos clonar de los 4 objetos. Después crear laberintos de otro tipo como LaberintosConBombas utilizar para ellos los prototipos de HabitacionConBomba y ParedExplosionada (es necesario implementar también en estas clases el método clonar), serán necesario constructor copia y algún método de inicialización para que los clientes puedan inicializar los prototipos. Implementar también un cliente que cree un laberinto de cada tipo.

- 6) **Patrón Builder:** Utilizando un Builder. Construir un coche por partes: Ruedas, Motor, Chasis. Tendremos dos builder distintos que construyen un tipo distinto de coche.

```
Jeep
body:SUV
engine horsepower:400
tire size:22'

Nissan
body:hatchback
engine horsepower:85
tire size:16'
```

- 7) **BUILDER:** A partir del proyecto LABERINTO. Definir un ConstructorLaberinto con la siguiente interfaz:

```
class ConstructorLaberinto {

protected:

    ConstructorLaberinto () {};
```

```

public:
    virtual void construirLaberinto() {}
    virtual void construirHabitacion(int habitacion) {}
    virtual void construirPuerta(int h1, int h2) {}

    virtual Laberinto *obtenerLaberinto() { return 0; }
};

```

(Al ser los métodos NO virtuales puros, las subclases pueden sobrescribir, los métodos que les interesen)

Se pretender tener dos subclases de esta:

ConstructorLaberintoEstandar y ConstructorLaberintoContador.

Definirlas, la 1ª construye el mismo laberinto de siempre, la otra se dedica a contar el número de puertas y habitaciones que añadimos.

Reescribir el método crearLaberinto de la clase Juego, ahora recibirá un constructor por parámetro y devuelve el Laberinto creado.

Escribir también un Cliente que haga uso de ambos constructores.

¿Cuáles son los participantes?

- 8) **Patrón Prototype Figuras editor grafico:** Diseñar un patrón prototype para almacenar los prototipos de figuras en 2D: Triangulo, Circulo y Rectángulo. Se seleccionarán por medio de un campo enumerado. Todas estas clases heredan de la clase abstracta Figura que mantiene una propiedad color y al menos los métodos (draw y clone). Ambos métodos se implementarán en las subclases. El método draw imprime las propiedades de cada Figura y el método clone devuelve el prototipo clonado. Posteriormente se diseñará una clase Escena que mantiene un vector de Figuras y que define operaciones para imprimir (draw) todas las figuras de la escena y add para añadir una figura a la escena. Esta clase utilizará la factoría de prototipos.

- 9) **Patrón Prototype:** Disponemos de distintos prototipos (CarRecord, BikeRecord, PersonRecord) partiendo de un prototipo genérico: Record. Todos los prototipos deben sobrescribir el método print y clone de la clase Record (serán métodos virtuales puros). Después dispondremos de una clase factoría donde almacenamos todos los prototipos y los podremos clonar. La factoría nos devuelve nuevos objetos que ha clonado a partir de los prototipos previamente creados.

```
Car Record
Name : Ferrari
Number: 5050

Bike Record
Name : Yamaha
Number: 2525

Person Record
Name : Tom
Age : 25
```

## PATRONES ESTRUCTURALES:

- 10) **Patrón Adapter:** se dispone de una librería de vectores tridimensionales implementados con una clase Vector3D (se encuentra en la carpeta de material):

```
#ifndef VECTOR3D_H_
#define VECTOR3D_H_

class Vector3D {
private:
    double x, y, z;
public:
    Vector3D();
    Vector3D(double, double, double);
    inline double getX() const { return x; }
    inline double getY() const { return y; }
    inline double getZ() const { return z; }
    double productoEscalar(const Vector3D &);
    double norma();
    virtual ~Vector3D();
};

#endif /* VECTOR3D_H_ */
```

Necesitamos una clase VectorPlano que implemente el interfaz:

```
#ifndef VECTOR2D_H_
#define VECTOR2D_H_

class Vector2D { // Es una clase Abstracta.

public:

    Vector2D(){}

    virtual double getAbcisa() const =0; // eje x

    virtual double getOrdenada()const =0; // eje y

    virtual double prod(const Vector2D &)=0; // producto escalar

    virtual double magnitud()=0; // norma

    virtual ~Vector2D(){};

};

#endif /* VECTOR2D_H_ */
```

La nueva clase tiene que adaptar la funcionalidad de Vector3D a Vector2D

Desde un cliente crear objetos VectorPlano y probar las funcionalidades de la clase. Implementar dos versiones de VectorPlano (por Composición y H.Múltiple).

11) **Patrón Fachada:** Implementar este patrón para simular la concesión de una Hipoteca. El proceso para el cliente es el siguiente: hay que comprobar si en el banco hay saldo por encima de un valor. En el sistema de créditos hay que comprobar si hay suficiente crédito y comprobar que el cliente no tenga impagos en el servicio de impagos.

12) **Patrón Decorator:** A partir de una clase Ventana que representa una ventana simple. Definir una serie de decoradores que permitan crear ventanas con Borde, con Botón de Ayuda, y las combinaciones de ambas. O una ventana con varios bordes. Se simulará con la consola de la siguiente forma: (el borde se representa con el carácter '|', y el botón con **[Botón de Ayuda]**)

```

      Ventana
    Ventana [Botón de Ayuda]
| Ventana [Botón de Ayuda]|
| Ventana |
|| Ventana ||
```

13) **Patrón Proxy**: Disponemos de un servidor real, con los atributos: host y puerto. Así como un método descargar que recibe un string con la URL que queremos descargar. El proxy mantiene los mismos atributos y métodos (más un método que comprueba si la URL está restringida o no).

Ambas clases heredan de una interface Servidor con los métodos comunes del ServidorReal y el Proxy. El Cliente (main) utiliza el ServidorProxy.

La salida del programa puede ser algo así:

```
Proxy inicializado ...
El proxy intercepta la url: prueba.doc
Servidor iniciado ...
Descargando http://free.es/80/prueba.doc
```

14) **Patrón Composite**: Se quiere construir un editor de expresiones matemáticas. Especificar el diagrama de clases que permita representar expresiones válidas. Una expresión válida estará formada o bien por un número, o bien por la suma/resta/división/multiplicación de dos expresiones. Ejemplos de expresiones válidas:

4      3+8      14 \* (3+5)

El programa mostrará lo siguiente:

```
( 14 - ( 3 * 5 ))
Resul : -1
```

15) **Patrón Bridge**: Implementar la jerarquía de **Documentos (Pedido y Horario)** junto con la de **Formatos (HTML y CSV)**. Los pedidos tienen los datos del **cliente**: cif, razón social y dirección. Y una serie de **detalles**: producto, cantidad y precio. Y el **Horario** una cabecera con un número de filas. Ejemplos de salida:

<b>PEDIDO:</b> <b>En HTML:</b> <pre> &lt;tr&gt; &lt;td&gt;8484833-A&lt;/td&gt; &lt;td&gt;Logista&lt;/td&gt; &lt;td&gt;Trigo 39&lt;/td&gt; &lt;/tr&gt; &lt;tr&gt; &lt;td&gt;portatil&lt;/td&gt; &lt;td&gt;2&lt;/td&gt; &lt;td&gt;550.000000&lt;/td&gt; &lt;td&gt;1100.000000&lt;/td&gt; &lt;/tr&gt; &lt;tr&gt; &lt;td&gt;mesa&lt;/td&gt; &lt;td&gt;1&lt;/td&gt; &lt;td&gt;120.000000&lt;/td&gt; &lt;td&gt;120.000000&lt;/td&gt; &lt;/tr&gt; </pre> <b>En CSU:</b> 8484833-A;Logista;Trigo 39 portatil;2;550.000000;1100.000000 mesa;1;120.000000;120.000000	<b>HORARIO:</b> <b>En HTML:</b> <pre> &lt;tr&gt; &lt;td&gt;L&lt;/td&gt; &lt;td&gt;M&lt;/td&gt; &lt;td&gt;X&lt;/td&gt; &lt;td&gt;J&lt;/td&gt; &lt;td&gt;U&lt;/td&gt; &lt;/tr&gt; &lt;tr&gt; &lt;td&gt;C++&lt;/td&gt; &lt;td&gt;Python&lt;/td&gt; &lt;td&gt;PHP&lt;/td&gt; &lt;td&gt;Java&lt;/td&gt; &lt;td&gt;C#&lt;/td&gt; &lt;/tr&gt; </pre> <b>En CSU:</b> L;M;X;J;U C++;Python;PHP;Java;C#
---	---

## PATRONES DE COMPORTAMIENTO

16) **Patrón Estrategia:** Crear 3 estrategias que implementen una interface y un contexto que será el encargado de seleccionar la estrategia aplicar. En este caso podemos utilizar los métodos de ordenación de un array. Ver carpeta de prácticas.

17) **Patrón State:** Modelizar los estados de un coche. La posible salida:

```

El vehiculo esta apagado
Velocidad actual: 0 Combustible: 50
El vehiculo ahora se encuentra parado
El vehiculo se encuentra ahora EN MARCHA
Velocidad actual: 10 Combustible: 40
Velocidad actual: 20 Combustible: 30
Velocidad actual: 30 Combustible: 20
Velocidad actual: 40 Combustible: 10
Velocidad actual: 20 Combustible: 0
El vehiculo se ha quedado sin combustible
Velocidad actual: 20 Combustible: 0

```

### Estados:

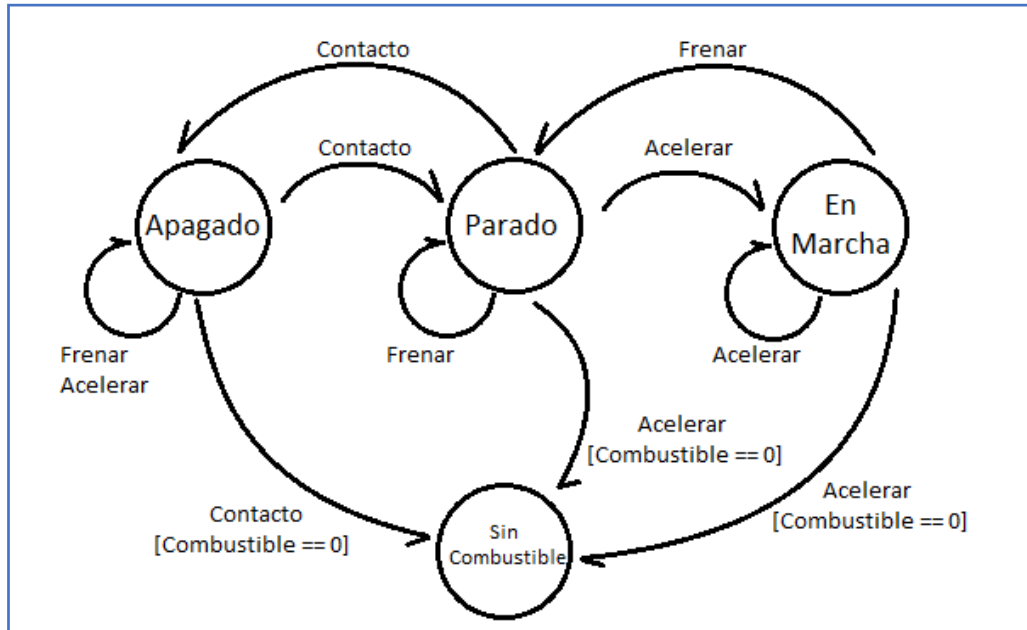
- Parado (pero con motor en marcha),
- En Marcha (en movimiento),
- Sin combustible
- Apagado

### Métodos:

- acelerar,
- frenar,
- contacto.

**El vehículo tiene 2 propiedades:**

- Velocidad (aumenta o disminuye de 10 en 10, según se acelere o se frene)
- Combustible (cada vez que acelera disminuye en 10).



18) **Patrón Cadena de Responsabilidad:** Definir una cadena de responsabilidad para procesar una petición. La petición se define a partir de un contenido (texto) y un identificador (utilizar una enum para los distintos tipos de petición). Cada manejador (de la petición) puede enviar la petición por un canal distinto (SMS, email, whatsapp). Se trata de implementar los 3 manejadores. El cliente (main) creará la cadena de responsabilidad y le enviará la petición al primero. Cada manejador lee el identificador de la petición y si es para él la procesa (y la envía por su canal) si no la pasa al sucesor.

Si la cadena es: WhatsApp → EMail → SMS

Y mandamos una petición para enviarla por WhatsApp la traza del programa es:

```

WhatsApp reenvia la peticion ...
Email reenvia la peticion ...
SMS: Mensaje de prueba 0
  
```

19) **Patrón Observer:**

A partir de este código que se encuentra en practicas/Tiempo/..

```

#include <stdio.h>
#include <time.h>
#include <string>
#include <iostream>
  
```



```

#include <windows.h>
using namespace std;

string getDateTime(){
    time_t rawtime;
    struct tm *timeinfo;

    time ( &rawtime );
    timeinfo = localtime ( &rawtime );
    string resul = asctime (timeinfo);
    return resul;
}

void retardo(int segundos){
    Sleep(1000);
}

int main (){
    for (int i = 0 ; i < 5 ; i++){
        cout << getDateTime() << endl;
        retardo(1);
    }

    return 0;
}

```

Se implementarán las siguientes clases:

- **Observador** (abstracta) con el método virtual puro, **actualizar(Sujeto)**
- **Sujeto**, mantiene la **lista de observadores**, permite operaciones de **añadir** un observador, y **notificar** a todos los observadores (con esta operación indicará que se ha producido un cambio).
- **Reloj** que hereda de **Sujeto** y emite un pulso cada sg. (se hará con el código que se proporciona).
- Los **dos observadores: RelojAnalogico** y **RelojDigital** pintan por pantalla un reloj.

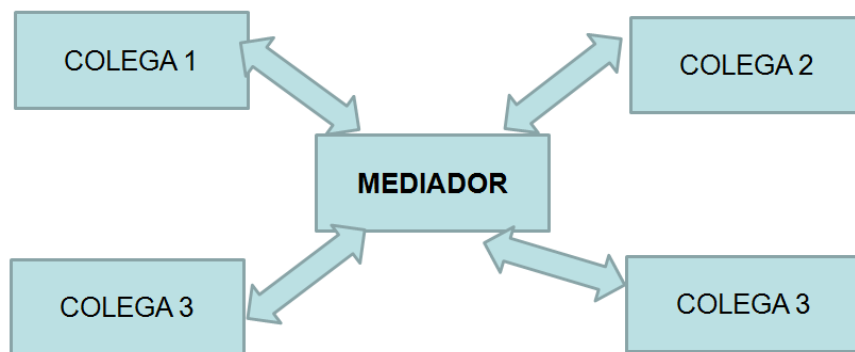
- 20) **Patrón Interpreter:** Implementar un patrón interpreter que sea capaz de Evaluar expresiones Boolean con al menos las operaciones: And, Or, Not, y que admita constantes con el valor: True / False. Será necesario una clase abstracta que disponga del método **evaluar()** y que devuelva el valor de la expresión. Cada expresión tiene que tener un método **toString** para pintarla por pantalla.

```

< F OR < < T AND < F OR T > > AND NOT < T > > >
Resul: FALSE

```

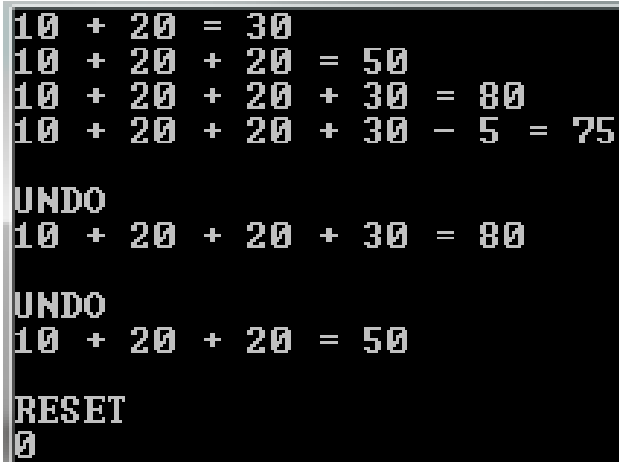
- 21) **Patrón Template Method:** Definir un patrón Template Method para poder realizar operaciones con un empleado (id, nombre, empresa, sueldo) . Serán operaciones de grabar, borrar y listar suponiendo destinos distintos (Fichero / Base de datos). Cada clase / método emitirá mensajes de donde esta realizando la operación, en fichero o en base de datos.
- 22) **Patrón Mediator:** Implementar el siguiente patrón de comunicación. Cada colega mantiene una referencia al mediador y cuando quiere enviar un mensaje a otro colega lo hace a través de mediador. Diseñar una clase Colega que tendrá métodos para inyectar el mediador y métodos para comunicar un mensaje y otro para recibir. El método comunicar de Colega utiliza el mediador para enviar su mensaje. Y el método recibir será específico de cada colega, deberá informar del mensaje recibido y de qué tipo de colega se trata (implementar al menos 2 tipos de colegas distintos). El mediador tiene que mantener una lista con todos los colegas que se comunican con él. En el main, se crean los colegas, se crea el mediador, se instalan los colegas en el mediador. Y un colega emite un mensaje y lo recibirán el resto de colegas (es una comunicación de 1 a muchos).



- 23) **Patrón Iterator:** Implementar un patrón iterator. Definir un vector que encapsula un array dinámico (se pueden utilizar templates) , tendrá un método que devuelva el iterator y esta clase dispondrá de los métodos (bool hasNext() y int next() o T next())
- 24) **Patrón Command:** Disponemos de un objeto: Light con dos métodos on / off para encender la luz. Por otro lado el interface Command dispone de un método: virtual void execute()=0. Se trata de implementar una clase Switch que se apoya en dos Command para encender y apagar la luz. Si pulsamos el interruptor la luz se enciende / apaga la Luz.
- 25) **Patrón Command + Memento:** Diseñar una calculadora que tiene 3 tipos de operaciones: operar(Operación \*) -> podrá ser Suma o Resta, Reset(): inicializa a 0 la calculadora. Undo(): deshace la última operación. En este caso habrá que hacer uso del patrón Memento para llevar un historial de las operaciones. La calculadora se inicializa a un valor y a partir de este, se le van aplicando sumas o restas:

**Ejemplo de uso:**

```
ICalculadora *calc = new Calculadora(10);  
calc->operar(new OperacionSuma(20));  
calc->operar(new OperacionSuma(20));  
calc->operar(new OperacionSuma(30));  
calc->operar(new OperacionResta(5));  
//calc->verHistorialDeComandos();  
calc->undo();  
calc->undo();  
calc->reset();  
delete calc;
```

**Salida del programa:**

```
10 + 20 = 30  
10 + 20 + 20 = 50  
10 + 20 + 20 + 30 = 80  
10 + 20 + 20 + 30 - 5 = 75  
  
UNDO  
10 + 20 + 20 + 30 = 80  
  
UNDO  
10 + 20 + 20 = 50  
  
RESET  
0
```

**En la carpeta: practicas/Memento\_Command hay algún fichero.**