UNIVERSIDAD VERACRUZANA

Facultad de Estadística e Informática

PATRONES DE DISEÑO

PATRONES DE CREACIÓN Y ESTRUCTURALES

15 DE FEBRERO DE 2017

ASIGNATURA: TÓPICOS SELECTOS DE COMPUTACIÓN I

PROFESOR: LUIS G. MONTANÉ-JIMÉNEZ

BLOQUE 8 SECCIÓN 2

ALUMNOS:

LEOBARDO CHIMAL HERNANDEZ

MAIRA LÓPEZ QUIROZ

MARYCRUZ MARTÍNEZ RAMOS

AMIZADAY MENDOZA

EMMANUEL MOLINA ROMERO

JESÚS EMILIO MONTIEL ESLAVA

INDICE

Patrones de Diseño

Los patrones ayudan a construir sobre la experiencia colectiva de ingenieros de software experimentados. Estos capturan la experiencia existente y que ha demostrado ser exitosa en el desarrollo de software, además ayudan a promover las buenas prácticas del diseño. Cada patrón aborda un problema específico y recurrente en el diseño o implementación de un sistema de software.

Los patrones de diseño son soluciones para problemas típicos y recurrentes que nos podemos encontrar a la hora de desarrollar una aplicación.

Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software. Brinda una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Debemos tener presente los siguientes elementos de un patrón: Nombre del patrón, sinopsis, contexto, causas, solución de estructura y estrategias, consecuencias, implementación y patrones relacionados.

Existen varios patrones de diseño, los cuales se clasifican en:

- **Patrones de creación:** relativo a la creación de objetos.
- **Patrones estructurales:** trata de la composición de clases u objetos.
- **Patrones de comportamiento:** caracteriza la forma en que clases y objetos interactúan y la distribución de las responsabilidades.

En los patrones de creación se encuentran:

1) Abstract Factory
2) Builder
3) Factory method
4) Prototype
5) Singleton

En los patrones estructurales se encuentran:

1) Adapter
2) Bridge
3) Composite
4) Decorator
5) Facade
6) Flyweight
7) Proxy

Y en los patrones de comportamiento se encuentran:

1) Chain of responsibility
2) Command
3) Interpreter
4) Iterator
5) Mediator
6) Memento
7) Observer
8) State
9) Strategy
10) Template method
11) Visitor

PATRONES DE CREACIÓN

Son aquellos patrones que se ocupan del proceso de creación de objetos. Procuran independizar al sistema de cómo sus objetos son creados y/o representados.

Abstract Factory

Nombre del patrón: Abstract Factory

Sinopsis:
Permite crear, mediante una interfaz, conjuntos o familias de objetos (denominados productos) que dependen mutuamente y todo esto sin especificar cuál es el objeto concreto. Interfaz para la creación de familias de objetos relacionados sin especificar sus clases concretas

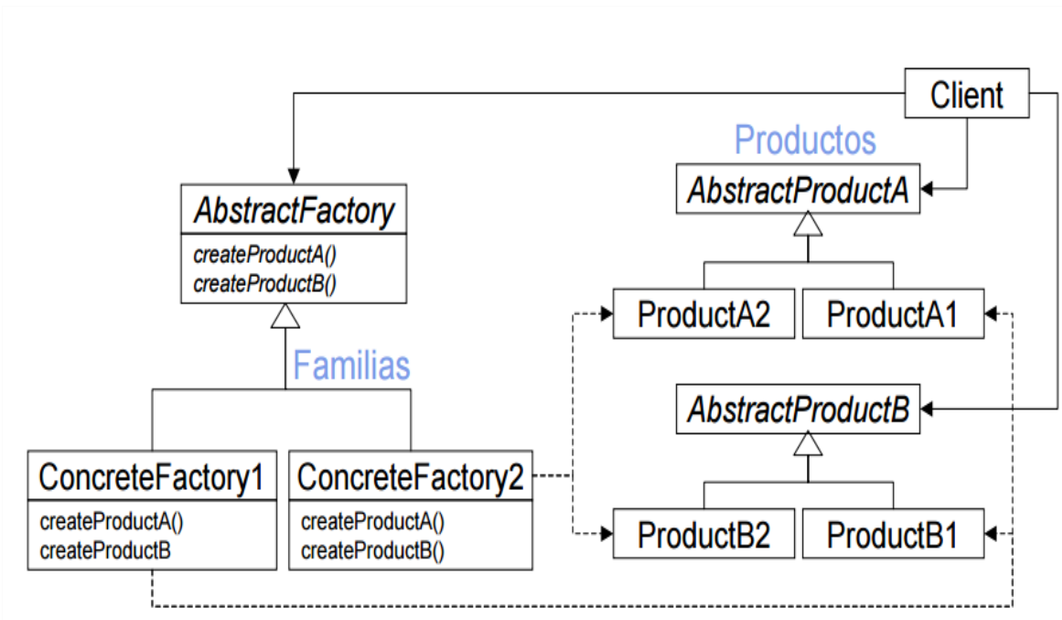
Contexto:
Un sistema debe ser independiente de la forma en que sus productos son creados, compuestos, y representados.
Un sistema debe ser configurado con una de muchas familias de productos disponibles
Unas familias de productos son diseñadas para su uso conjunto, y se requiere asegurar este uso conjunto.
Se desea proporcionar una biblioteca de productos presentando su interfaz, pero no su implementación.

Causas:
Por razones de portabilidad el resto del sistema no debe hacer referencia a componentes concretos.

Solución:

- Clase abstracta con un método de fabricación para cada componente
- Clase abstracta para cada componente
- El cliente crea componentes a través de la fábrica abstracta
- La fábrica de componentes garantiza el uso consistente de componentes de un mismo look-and feel

Estructura:



Donde:

- **AbstractFactory:** define la interfaz para crear objetos producto abstractos.
- **ConcreteFactory:** implementa las operaciones para crear objetos producto concretos
- **AbstractProduct:** define la interfaz de un tipo de objeto producto.
- **ConcreteProduct:** define un objeto producto a crear con la factoría concreta correspondiente z implementa la interfaz.
- **AbstractProductClient:** sólo usa las interfaces de AbstractFactory y AbstractProduc.

Estrategias:

1- Clase abstracta con un método de fabricación para cada componente.
2- El cliente crea componentes a través de la fábrica abstracta.

Consecuencias:

- Aísla clases concretas (no aparecen en el código del cliente).
- Facilita el intercambio de familias de productos.

Implementación:

Fabricas usualmente Instancia Única.
Creación de productos.
Método de Fabricación (normal).
Prototipo (menos habitual, aunque evita extender Fabrica Abstracta).
Flexibilizar fabrica mediante parametrización de método de fabricación.
Método de fabricación puede crear distintos tipos de componentes en base a los parámetros del método ◦ El cliente debe realizar una conversión tras crear el producto (downcasting).
(En la mayoría de los lenguajes) no hay ninguna relación formal entre valores de los parámetros y productos creados (salvo código fuente).

Java API:
Código:

Código de ejemplo: laberinto

```
// factoría abstracta (proporciona una implementación por defecto)
public class MazeFactory {
   Maze makeMaze () { return new Maze(); }
   Wall makeWall () { return new Wall(); }
   Room makeRoom (int n) { return new Room(n); }
   Door makeDoor (Room r1, Room r2) { return new Door(r1, r2); }
}
public class MazeGame {
   // @param MazeFactory factory: factoría a usar para la creación de componentes
   Maze createMaze (MazeFactory factory) {
      Maze aMaze = factory.makeMaze();
      Room r1 = factory.makeRoom(1), r2 = factory.makeRoom(2);
      Door aDoor = factory.makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(Direction.NORTH, factory.makeWall());
      r1.setSide(Direction.EAST,  aDoor);
      r1.setSide(Direction.SOUTH, factory.makeWall());
      r1.setSide(Direction.WEST,  factory.makeWall());
      r2.setSide(Direction.NORTH, factory.makeWall());
      r2.setSide(Direction.EAST,  factory.makeWall());
      r2.setSide(Direction.SOUTH, factory.makeWall());
      r2.setSide(Direction.WEST,  aDoor);
      return aMaze;
 }}
```

```
// factorías concretas (sobrescriben métodos de la factoría abstracta)

public class BombedMazeFactory extends MazeFactory {
   Wall makeWall ()      { return new BombedWall(); }
   Room makeRoom (int n) { return new RoomWithABomb(n); }
}

public class EnchantedMazeFactory extends MazeFactory {
   Room makeRoom (int n) { return new EnchantedRoom(n, castSpell());}
   Door makeDoor (Room r1, Room r2) {return new DoorNeedingSpell(r1, r2);}
   protected Spell castSpell() { }
}

// cliente

public class MazeTest {
   public static void main (String args[]) {
      MazeGame game = new MazeGame();
      MazeFactory factory = new BombedMazeFactory();
      game.createMaze(factory);
   }
}
```

Patrones relacionados:
Singleton

Builder

Nombre del Patrón: Builder

Sinopsis:

Es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder.

El patrón Builder es creacional.

A menudo, el patrón Builder construye el patrón Composite, un patrón estructural.

Abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

Contexto:

Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones. Simplifica la construcción de objetos con estructura interna compleja y permite la construcción de objetos paso a paso.

Un único proceso de construcción debe ser capaz de construir distintos objetos complejos, abstrayéndonos de los detalles particulares de cada uno de los tipos.

Causas:

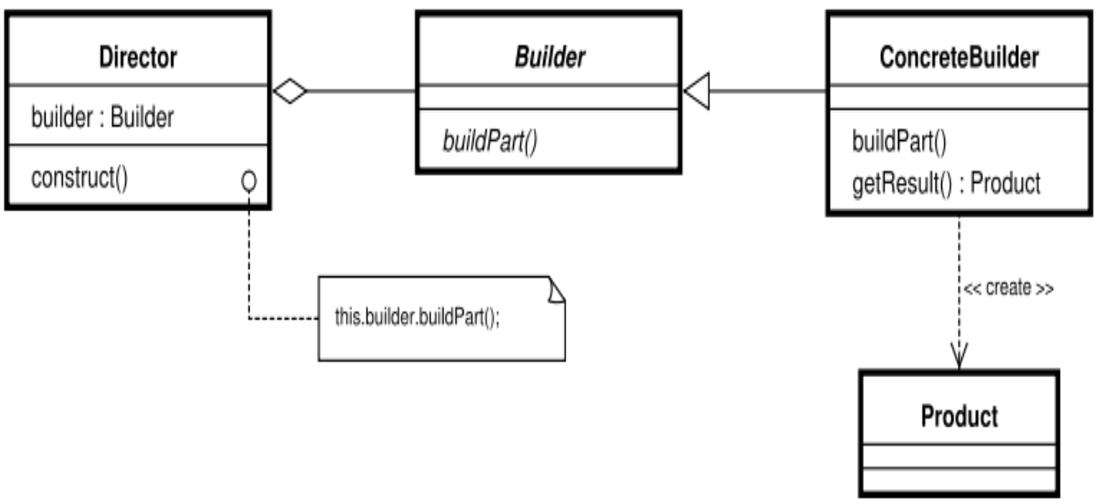
Nuestro sistema trata con objetos complejos (compuestos por muchos atributos) pero el número de configuraciones es limitado.

El algoritmo de creación del objeto complejo puede independizarse de las partes que lo componen y del ensamblado de las mismas.

Solución:

La solución será crear un constructor que permita construir todos los tipos de objetos, ayudándose de constructores concretos encargados de la creación de cada tipo en particular. Un objeto director será el encargado de coordinar y ofrecer los resultados.

Estructura:



Donde:

Director: Se encarga de construir un objeto utilizando el Constructor (Builder).

Builder: Interfaz abstracta que permite la creación de objetos.

Concrete Builder: Implementación concreta del Builder definida para cada uno de los tipos. Permite crear el objeto concreto recopilando y creando cada una de las partes que lo compone.

Product: Objeto que se ha construido tras el proceso definido por el patrón.

Consecuencias:

Positivas:

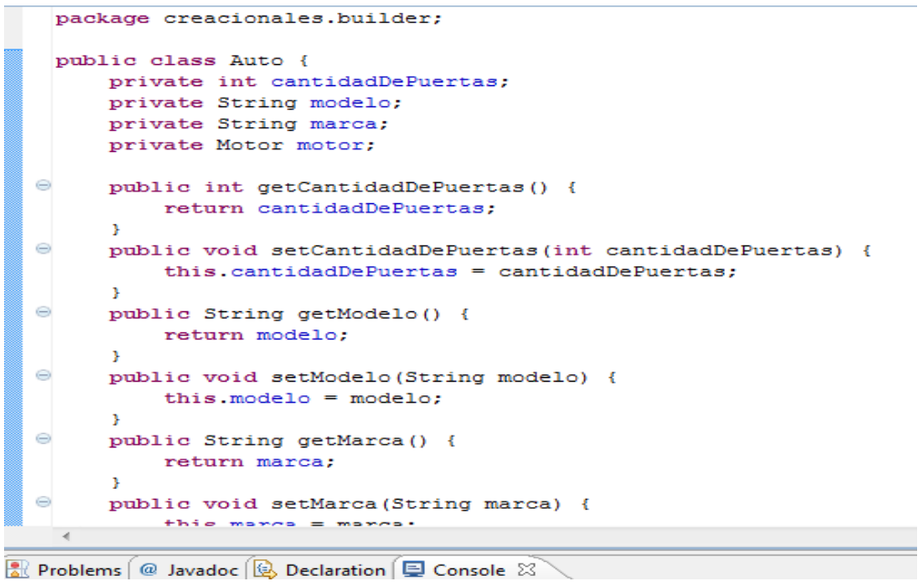
- Reduce el acoplamiento.
- Permite variar la representación interna del objeto, respetando la clase builder. Es decir, conseguimos independizar la construcción de la representación.

Negativas:

- Introduce complejidad en los programas.

Implementación:

El objetivo del ejemplo es poder crear un objeto Auto (este sería nuestro producto). El auto se compondrá de varios atributos que lo componen: motor, marca, modelo y cantidad de puertas. En nuestro ejemplo, el auto no se compone de muchos objetos complejos. De hecho, se compone de sólo 4 objetos relativamente sencillos. Esto es para poder hacer entendible la propuesta del Builder y no perderse en los objetos que lo componen.



```java
package creacionales.builder;

public class Auto {
    private int cantidadDePuertas;
    private String modelo;
    private String marca;
    private Motor motor;

    public int getCantidadDePuertas() {
        return cantidadDePuertas;
    }
    public void setCantidadDePuertas(int cantidadDePuertas) {
        this.cantidadDePuertas = cantidadDePuertas;
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo(String modelo) {
        this.modelo = modelo;
    }
    public String getMarca() {
        return marca;
    }
    public void setMarca(String marca) {
```

Problems | Javadoc | Declaration | Console

```java
package creacionales.builder;

public class Motor {
    private int numero;
    private String potencia;

    public int getNumero() {
        return numero;
    }
    public void setNumero(int numero) {
        this.numero = numero;
    }
    public String getPotencia() {
        return potencia;
    }
    public void setPotencia(String potencia) {
        this.potencia = potencia;
    }

}
```

Siguiendo con nuestro ejemplo, definimos nuestro Builder llamado AutoBuilder. El Builder define al menos dos cosas: un método para devolver el Producto (el auto en nuestro caso) y los métodos necesarios para la construcción del mismo.

Serán los ConcreteBuilders los encargados de colocarle la lógica de construcción de cada Auto en particular. En nuestro caso, tendremos dos ConcreteBuilder: FiatBuildery FordBuilder. Recordemos que, en nuestro ejemplo, son clases que construyen objetos muy sencillos con datos hardcodeados para facilitar el aprendizaje del patrón en sí.

```java
package creacionales.builder;

public abstract class AutoBuilder {
    protected Auto auto;

    public Auto getAuto() {
        return auto;
    }

    public void crearAuto() {
        auto = new Auto();
    }

    public abstract void buildMotor();

    public abstract void buildModelo();

    public abstract void buildMarca();

    public abstract void buildPuertas();
}
```

```java
package creacionales.builder;

public class FiatBuilder extends AutoBuilder {

    public void buildMarca() {
        auto.setMarca("Fiat");
    }

    public void buildModelo() {
        auto.setModelo("Palio");
    }

    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(232323);
        motor.setPotencia("23 HP");
        auto.setMotor(motor);
    }

    public void buildPuertas() {
        auto.setCantidadDePuertas(2);
    }
}
```

```java
package creacionales.builder;

public class FordBuilder extends AutoBuilder {

    public void buildMarca() {
        auto.setMarca("Ford");
    }

    public void buildModelo() {
        auto.setModelo("Focus");
    }

    public void buildMotor() {
        Motor motor = new Motor();
        motor.setNumero(21212);
        motor.setPotencia("20 HP");
        auto.setMotor(motor);
    }

    public void buildPuertas() {
        auto.setCantidadDePuertas(4);
    }
}
```

Por último, realizaremos el Director. Lo primero que debe hacerse con esta clase es enviarle el tipo de auto que se busca construir (Ford, Fiat, etc.). Luego, al llamar al método constructAuto(), la construcción se

realizará de manera automática.

```
package creacionales.builder;

public class AutoDirector {
    // No es necesario que exista la palabra Director
    // Esta clase podría llamarse Concesionaria, Garage, FabricaDeAutos, etc

    private AutoBuilder autoBuilder;

    public void constructAuto() {
        autoBuilder.crearAuto();
        autoBuilder.buildMarca();
        autoBuilder.buildModelo();
        autoBuilder.buildMotor();
        autoBuilder.buildPuertas();
    }

    public void setAutoBuilder(AutoBuilder ab) {
        autoBuilder = ab;
    }

    public Auto getAuto() {
        return autoBuilder.getAuto();
    }
}
```

La invocación desde un cliente sería

```
package creacionales.builder;

public class Main {

    public static void main(String[] args) {
        AutoDirector director = new AutoDirector();
        director.setAutoBuilder(new FordBuilder());
        director.constructAuto();
        Auto auto = director.getAuto();

        System.out.println(auto.getMarca());
        System.out.println(auto.getModelo());
    }

}
```

Problems | Javadoc | Declaration | Console

<terminated> Main (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (

```
Ford
Focus
```

:

Patrones Relacionados: Método Factoría (Factory Method).

FACTORY METHOD

Nombre: Factory method

Sinopsis:

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Contexto:

El problema que se plantea en algunos entornos es que una clase no puede anticipar el tipo de objetos que debe crear debido a la jerarquía de clases existente, lo cual provoca que tenga que delegar esta tarea en una subclase. En otras palabras, viene a solucionar el problema que se presenta cuando tenemos que crear la instancia de un objeto, pero a priori no sabemos aún que tipo de objeto tiene que ser, generalmente, porque depende de alguna

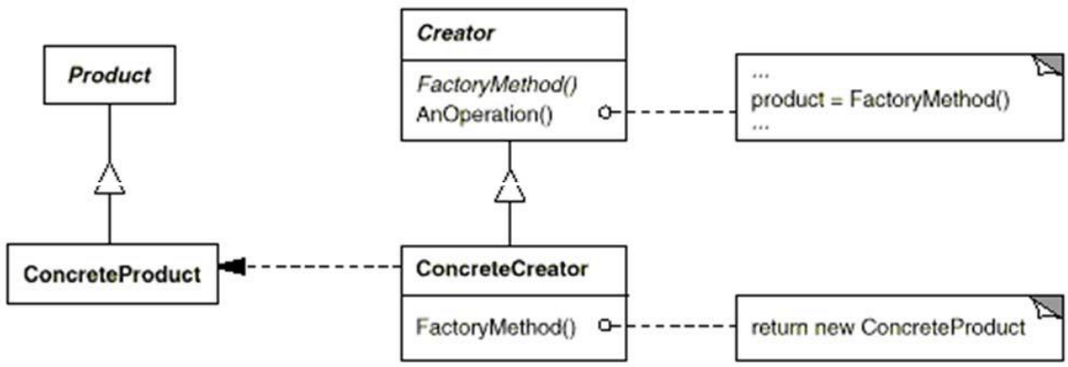
opción que seleccione el usuario en la aplicación o porque depende de una configuración que se hace en tiempo de despliegue de la aplicación.

Causas:

- No se puede anticipar la clase de objetos que se tienen que crear.
- Una clase quiere que sus subclases especifiquen que objetos se deben crear.
- Clases que delegan la responsabilidad a una de las subclases colaboradoras y se desea localizar el conocimiento de qué subclases es la encargada.

Solución:

Estructura: Estructura básica de la solución.



Participantes:

Product (Documento) - Define la interfaz de los objetos que crea el método de fabricación (método factoría).
ConcreteProduct (MiDocumento) - Implementa la interfaz Producto.
Creator (Aplicación)- Declara el método de fabricación, el cual devuelve un objeto del tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelve un objeto ProductoConcreto. Puede llamar al método de fabricación para crear un objeto Producto.
CreadorConcreto (MiAplicacion) - Redefine el método de fabricación para devolver una instancia de ProductoConcreto.
Colaboraciones entre participantes: El creador emplea el método de fabricación redefinido por sus subclases para utilizar la instancia del producto concreto apropiada.

Estrategias:

- **C**uando una clase no puede adelantar las clases de objetos que debe crear.
- Cuando una clase pretende que sus subclases especifiquen los objetos que ella crea.
- Cuando una clase delega su responsabilidad hacia una de entre varias subclases auxiliares y queremos tener localizada a la subclase delegada.

Consecuencias:

- Elimina la necesidad de incluir clases específicas de la aplicación en código que trata con la interfaz Product y funciona con cualquier clase ConcreteProduct.
- Mayor flexibilidad dado que se proporciona un mecanismo a las subclases para introducir una versión más extendida del producto.
- Conecta jerarquías de clases paralelas (delegación).
- Proporciona enganches para las subclases.
- La creación de objetos con métodos factoría es más flexible.
- Las subclases pueden dar una versión extendida del código padre.

Desventaja:

Puede obligar a extender la clase creadora sólo para crear un producto concreto. Si esto es un problema puede emplear otra solución como por ejemplo el patrón prototipo.

Implementación:

Dos variantes:

- El creador es una clase abstracta y no proporciona implementación por defecto para el método (necesario extender al creador)
- El creador es una clase concreta y define un producto concreto por defecto (flexibilidad para cambios futuros).

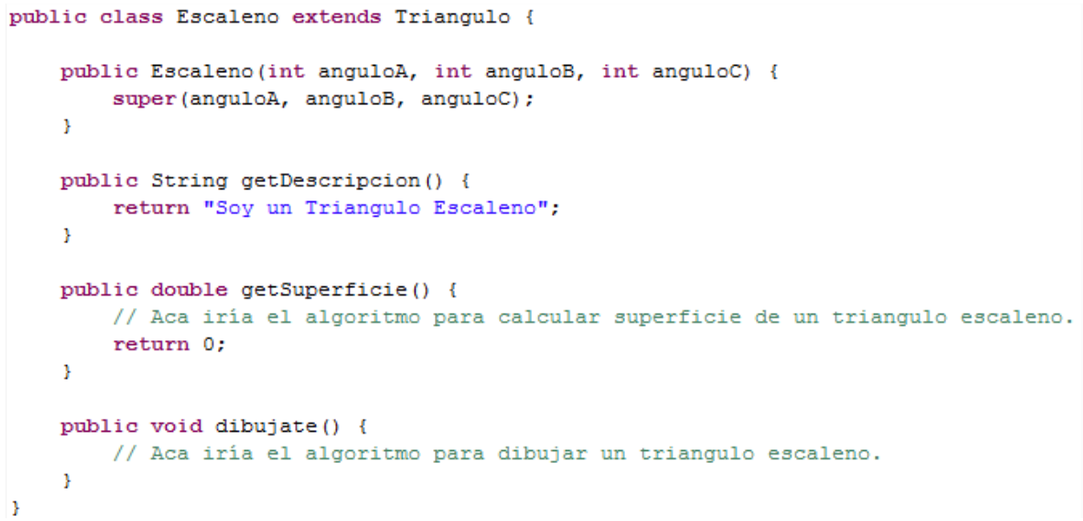
Métodos de fabricación parametrizados:

Un único método puede crear distintos productos en base a los parámetros del método de fabricación. (Solución más común).

Ejemplo: En nuestro ejemplo tenemos una clase abstracta llamada Triangulo, de la cual heredan los 3 tipos de triángulos conocidos.

```
public abstract class Triangulo {
    private int ladoA;
    private int ladoB;
    private int ladoC; // con sus get y set

    public Triangulo(int ladoA, int ladoB, int ladoC) {
        setLadoA(ladoA);
        setLadoB(ladoB);
        setLadoC(ladoC);
    }

    //Cada subclase debe redefinir estos tres métodos abstractos.
    public abstract String getDescripcion();

    public abstract double getSuperficie();

    public abstract void dibujate();

    public int getLadoA() {
        return ladoA;
    }

    public void setLadoA(int ladoA) {
```

```
public class Equilatero extends Triangulo {

    public Equilatero(int anguloA, int anguloB, int anguloC) {
        super(anguloA, anguloB, anguloC);
    }

    public String getDescripcion() {
        return "Soy un Triangulo Equilatero";
    }

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo equilatero.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo equilatero.
    }
}
```

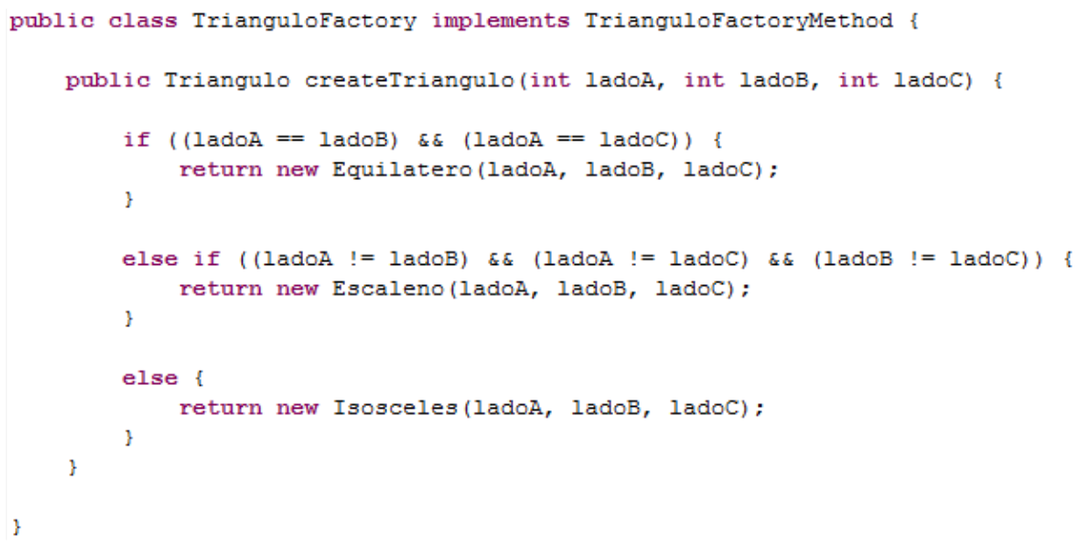


```
public class Escaleno extends Triangulo {

    public Escaleno(int anguloA, int anguloB, int anguloC) {
        super(anguloA, anguloB, anguloC);
    }

    public String getDescripcion() {
        return "Soy un Triangulo Escaleno";
    }

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo escaleno.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo escaleno.
    }
}
```

```
public class Isosceles extends Triangulo {

    public Isosceles(int ladoA, int ladoB, int ladoC) {
        super(ladoA, ladoB, ladoC);
    }

    public String getDescripcion() {
        return "Soy un Triangulo Isosceles";
    }

    public double getSuperficie() {
        // Aca iría el algoritmo para calcular superficie de un triangulo isosceles.
        return 0;
    }

    public void dibujate() {
        // Aca iría el algoritmo para dibujar un triangulo isosceles.
    }

}
```

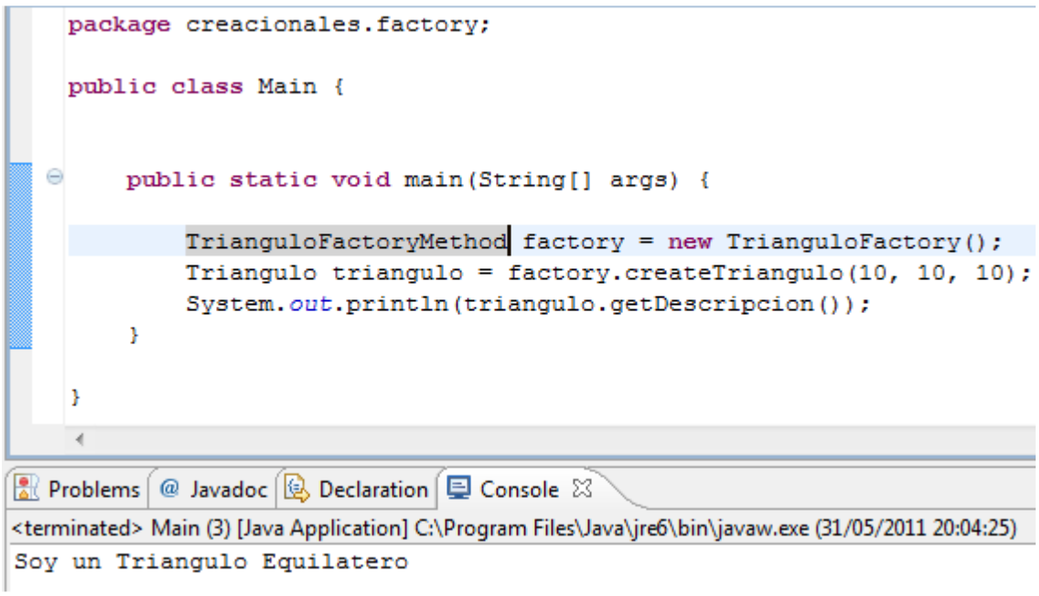
Pero tenemos el siguiente inconveniente: quien se encarga de crear un tipo de triangulo concreto no debería conocer cómo se compone internamente. Para ello hemos creado la clase TrianguloFactory con su correspondiente interface.

```
public interface TrianguloFactoryMethod {
    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC);
}
```



```
public class TrianguloFactory implements TrianguloFactoryMethod {

    public Triangulo createTriangulo(int ladoA, int ladoB, int ladoC) {

        if ((ladoA == ladoB) && (ladoA == ladoC)) {
            return new Equilatero(ladoA, ladoB, ladoC);
        }

        else if ((ladoA != ladoB) && (ladoA != ladoC) && (ladoB != ladoC)) {
            return new Escaleno(ladoA, ladoB, ladoC);
        }

        else {
            return new Isosceles(ladoA, ladoB, ladoC);
        }
    }

}
```

Veremos que, desde el punto de vista del cliente, es muy sencillo poder crear un triángulo:



Patrones relacionados:

- ***Abstract Factory:*** Este patrón suele implementarse con métodos de fabricación.
- ***Template Method:*** Los métodos que generalmente son llamados desde adentro del patrón son *Template Method*.
- ***Prototype:*** El creador necesita una operación para inicializar la clase producto.

PROTOTYPE

Nombre: Prototype

Sinopsis:

Su finalidad es crear nuevos objetos duplicándolos, clonando una instancia creada previamente.

Este patrón especifica la clase de objetos a crear mediante la clonación de un prototipo que es una instancia ya creada. La clase de los objetos que servirán de prototipo deberá incluir en su interfaz la manera de solicitar una copia, que será desarrollada luego por las clases concretas de prototipos.

Contexto:

El coste de crear un objeto nuevo desde 0 es muy elevado, y más aún si luego hay que establecer una gran colección de atributos. En éste contexto sería más conveniente clonar un objeto predeterminado que actúe de prototipo y modificar los valores necesarios para que se ajuste a su nuevo propósito.

Causas:

Para evitar las subclases de un objeto creador como hace el patrón Abstract Factory.

Para evitar el costo inherente a la creación de un objeto nuevo mediante el operador new cuando esto demasiado costoso para la aplicación.

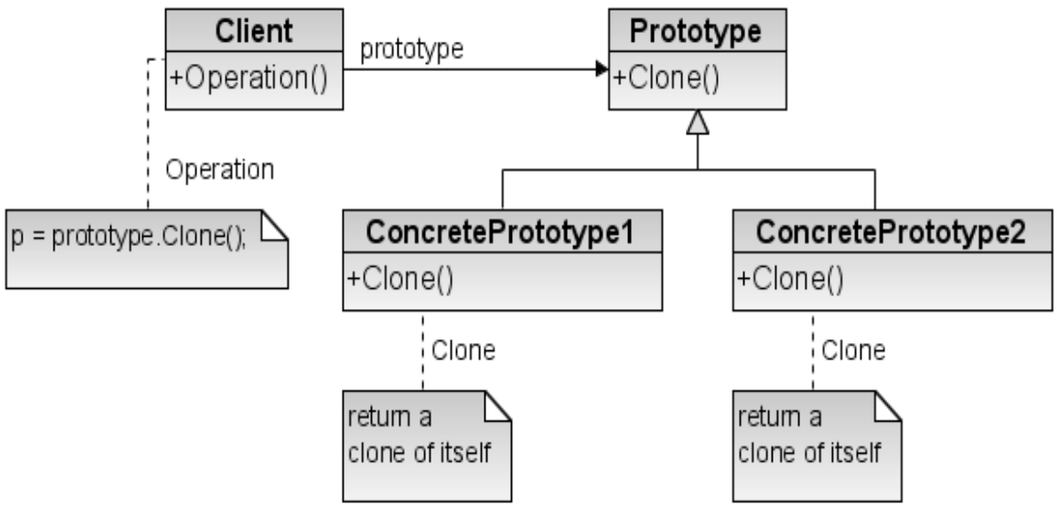
Solución:

Consistirá en definir una interfaz que expone el método necesario para realizar la clonación del objeto. Las clases que pueden ser clonadas implementarán esta interfaz, mientras que las clases que deseen clonar deberán utilizar el método definido en la interfaz.

Además, existen 2 tipos de clonación: La clonación profunda y la clonación superficial.

- En la clonación superficial modificar las referencias a terceros objetos hace que los originales cambien, ya que los terceros objetos son en realidad punteros.
- En la clonación profunda se clonan los terceros objetos dando lugar a nuevas referencias independientes.

La estructura es la siguiente:



Participantes:

- ***Cliente:*** Es el encargado de solicitar la creación de los nuevos objetos a partir de los prototipos.
- ***Prototipo Concreto:*** Posee características concretas que serán reproducidas para nuevos objetos e implementa una operación para clonarse.
- ***Prototipo:*** Declara una interfaz para clonarse, a la que accede el cliente

Consecuencias:

- **POSITIVAS:**
  - Clonar un objeto es mucho más rápido que crearlo.
  - Un programa puede añadir y borrar dinámicamente objetos prototipo en tiempo de ejecución.
  - El cliente no debe conocer los detalles de cómo construir los objetos prototipo.
- **NEGATIVAS**
  - En objetos muy complejos, implementar la interfaz Prototype puede ser muy complicada.

Desventajas:

La jerarquía de prototipos debe ofrecer la posibilidad de clonar un elemento y esta operación puede no ser sencilla de implementar. Por otro lado, si la clonación se produce frecuentemente, el coste puede ser importante.

Implementación:

Tenemos una fábrica de camisetas con estampados, típicas de las ferias y mercadillos. Para crear nuevas camisetas, cogeremos una similar y modificaremos únicamente el color, la talla y el estampado. Empezamos con el prototipo:

```
public abstract class Camiseta {
	private String nombre;
	private Integer talla;
	private String color;
	private String manga;
	private String estampado;
	private Object material;

	public Camiseta (String nombre,Integer talla, String color, String manga, String estampado, Object material){
		this.nombre = nombre;
		this.talla = talla;
		this.color = color;
		this.manga = manga;
		this.estampado = estampado;
		this.material = material;
	}
	public abstract Camiseta clone();
	/*
	 * Todos los getter y los setter.
	*/
}
```

Ahora construiremos los prototipos concretos para camisetas de manga larga y manga corta:

```
public class CamisetaMCorta extends Camiseta{
	public CamisetaMCorta(Integer talla, String color, String estampado){
		this.nombre = "Prototipo";
		this.talla = talla;
		this.color = color;
		this.manga = "Corta";
		this.estampado = estampado;
		this.material = new Lana();
	}
	public Camiseta clone(){
		return new CamisetaMCorta(this.talla, this.color, this.estampado);
	}
}


public class CamisetaMLarga extends Camiseta{
	public CamisetaMLarga(Integer talla, String color, String estampado){
		this.nombre = "Prototipo";
		this.talla = talla;
		this.color = color;
		this.manga = "Larga";
		this.estampado = estampado;
		this.material = new Lana();
	}
	public Camiseta clone(){
		return new CamisetaMLarga(this.talla, this.color, this.estampado);
	}
}
```

Por último, el método main hará de cliente y creará distintas camisetas tanto de manga larga como de manga corta a partir de prototipos.

```
public static void main(String[] args){
// Recibiremos en los argumentos los estampados de las camisetas

// Creamos los prototipos
Camiseta prototipoMCorta = new CamisetaMCorta(40, “blanco”, “Logotipo”);
Camiseta prototipoMLarga = new prototipoMLarga(40, “blanco”, “Logotipo”);

// Almacenamos las camisetas disponibles
ArrayList camisetas = new ArrayList();

for(int i = 0; i<args.length;i++){
                Camiseta cc =  prototipoMCorta.clone();
                cc.setEstampado(args[i]);

                for(int j = 35; j<60; j++){
                        Camiseta cc_talla = cc.clone();
                        cc_talla.setTalla(j);
                        camisetas.add(cc_talla);
                }


                Camiseta cl =  prototipoMLarga.clone();
                cl.setEstampado(args[i]);

                for(int j = 35; j<60; j++){
                        Camiseta cl_talla = cl.clone();
                        cl_talla.setTalla(j);
                        camisetas.add(cl_talla);
                }
        }
}
```

SINGLETON

Nombre del patrón: Singleton

Como curiosidad es interesante mencionar que la palabra singleton significa en inglés "un conjunto que contiene un solo miembro".

Sinopsis:

Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella
Contexto: Utilizado cuando se requiere exactamente una instancia de una clase. Es necesario el acceso controlado a un solo objeto.

Causas:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

Solución:
Garantizar una única instancia

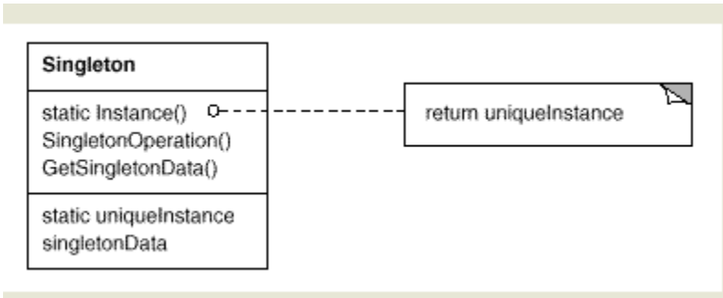


Diagrama OMT de Singleton, tomado del libro del GoF.

- **Participantes:** Define una operación Instancia que permite que los clientes accedan a su única instancia. Instancia es una operación de clase (static en C# y shared en VB .NET).
  Puede ser responsable de crear su única instancia.

  Colaboraciones: Los clientes acceden a una instancia de Singleton únicamente a través de Instancia de Singleton operación.

Consecuencias:
-El modelo Singleton tiene varios beneficios:
1. Acceso controlado a la instancia única. Debido a que la clase Singleton
Encapsula su única instancia, puede tener un control estricto sobre cómo y
Cuando los clientes lo acceden.

2. Espacio de nombres reducido. El patrón Singleton es una mejora frente a la globalización
Variables. Evita contaminar el espacio de nombres con variables globales que
Almacenar instancias únicas.

3. Permite refinamiento de operaciones y representación. La clase Singleton
Puede ser subclasificado, y es fácil configurar una aplicación con una instancia
De esta clase extendida. Puede configurar la aplicación con una instancia
De la clase que necesita en tiempo de ejecución.

4. Permite un número variable de instancias. El patrón facilita el cambio
Su mente y permitir más de una instancia de la clase Singleton. Además,
Puede utilizar el mismo enfoque para controlar el número de instancias que utiliza la aplicación. Sólo la operación que concede el acceso al Singleton
La instancia necesita cambiar.

5. Más flexible que las operaciones de clase. Otra forma de empaquetar un singleton
Es usar las operaciones de clase (es decir, las funciones de miembro estático
En C ++ o métodos de clase en Smalltalk). Pero ambas técnicas lingüísticas
Hacer que sea difícil cambiar un diseño para permitir más de una instancia de una clase.
Además, las funciones de miembro estático en C ++ nunca son virtuales, por lo que las subclases No pueden anularlos de forma polimórfica.

Implementación:
A continuación, se describen los problemas de implementación que deben tenerse en cuenta al utilizar el patrón Singleton:

1. Garantizar una instancia única. El patrón Singleton es la única instancia
Una clase normal de una clase, pero esa clase está escrita para que sólo una
Instancia se puede crear. Una forma común de hacerlo es ocultar la operación.
Que crea la instancia detrás de una operación de clase (es decir, una Miembro o un método de clase) que garantiza sólo una instancia es creado.
Esta operación tiene acceso a la variable que contiene
Ejemplo, y asegura que la variable se inicialice con la Instancia antes de devolver su valor. Este enfoque asegura que un Singleton Se crea e inicializa antes de su primer uso.
Puede definir la operación de clase en C ++ con una función miembro estática

Instancia de la clase Singleton. Singleton también define un miembro estático
Variable instance que contiene un puntero a su instancia única.
La clase Singleton se declara como:

```
class Singleton {
public:
static Singleton* Instance();
protected:
Singleton();
private:
static Singleton* _instance;
};
```

The corresponding implementation is

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
if (_instance == 0) {
_instance = new Singleton;
}
return _instance;
```

otro:

```
public class CSingleton {
private static CSingleton laInstancia = new CSingleton();
private CSingleton() {}
public static CSingleton getInstance() {
return laInstancia;
}
```

Explicación código:

Los clientes acceden al singleton exclusivamente a través del miembro Instance función. La variable instance se inicializa a 0, y el miembro estático La instancia de la función devuelve su valor, inicializándola con la Instancia si es 0. La instancia usa la inicialización perezosa; El valor que devuelve No se crea y almacena hasta que se accede por primera vez.
Observe que el constructor está protegido. Un cliente que intenta instanciar Singleton directamente obtendrá un error en tiempo de compilación. Esto asegura que Sólo se puede crear una instancia. Además, puesto que la inicialidad es un puntero a un objeto Singleton, La función de miembro de instancia puede asignar un puntero a una subclase de Singleton a esta variable. Vamos a dar un ejemplo de esto en el código de ejemplo.
Hay otra cosa a tener en cuenta sobre la implementación de C ++. No es suficiente para definir el singleton como un objeto global o estático y luego confiar en Inicialización automática. Hay tres razones para esto:

1. No podemos garantizar que sólo una instancia de un objeto estático ser declarado.
2. Podríamos no tener suficiente información para instanciar cada Singleton. En el momento de la inicialización estática. Un singleton puede requerir valores que se calculan posteriormente en la ejecución del programa.
3. C ++ no define el orden en el que los constructores de objetos globales se llaman a través de las unidades de traducción [ES90]. Esto significa que no pueden existir dependencias entre singletons; Si alguno hace, entonces los errores son inevitables.

Una responsabilidad adicional (aunque pequeña) del enfoque de objetos globales / estáticos es que obliga a todos los singletons a ser creado si se utilizan o no.
El uso de una función de miembro estático evita todos estos problemas. En Smalltalk, la función que devuelve la instancia única se implementa como un método de clase en la clase Singleton. Para asegurar que sólo una instancia se crea, anula la nueva operación. La clase Singleton resultante podría tienen los siguientes dos métodos de clase, donde SoleInstance es una variable de clase que no se utiliza en ningún otro lugar:

```
new
self error: 'cannot create new object'

default
SoleInstance isNil ifTrue: [SoleInstance := super new].
^ SoleInstance
```

Subclasificar la clase Singleton. El tema principal no es tanto definir la subclase pero instalando su instancia única para que los clientes sean capaz de usarlo. En esencia, la variable que se refiere al Singleton instancia debe

inicializarse con una instancia de la subclase. Lo más simple técnica consiste en determinar qué singleton desea utilizar en el Singleton funcionamiento de la instancia. Un ejemplo en el código muestra muestra cómo implementar esta técnica con variables de entorno.

Otra forma de elegir la subclase de Singleton es tomar la implementación de instancia fuera de la clase padre (por ejemplo, MazeFactory) y ponerlo en el subclase Eso permite que un programador C ++ decida la clase de singleton en Link-time (por ejemplo, enlazando en un archivo de objeto que contiene una implementación), pero lo mantiene oculto de los clientes del singleton.
El acercamiento del acoplamiento fija la opción de la clase del singleton en el tiempo del acoplamiento, que Hace que sea difícil elegir la clase singleton en tiempo de ejecución. Usando condicional declaraciones para determinar la subclase es más flexible, pero hard-wires el conjunto de posibles clases Singleton. Ninguno de los enfoques es suficientemente flexible en todos los casos.
Un enfoque más flexible utiliza un registro de singletons. En vez de tener instance define el conjunto de posibles clases Singleton, las clases Singleton pueden registrar su instancia singleton por su nombre en un registro bien conocido.
El registro mapea entre nombres de cadenas y singletons. Cuando la Instancia necesita un singleton, consulta el registro, pidiendo el singleton por su nombre.
El registro busca el singleton correspondiente (si existe) y devuelve eso. Este enfoque libera Instance de conocer todas las posibles clases Singleton o instancias. Todo lo que requiere es una interfaz común para todas las clases Singleton que incluye operaciones para el registro:

```
class Singleton {
public:
static void Register(const char* name, Singleton*);
static Singleton* Instance();
protected:
static Singleton* Lookup(const char* name);
private:
static Singleton* _instance;
static List<NameSingletonPair>* _registry;
};
```

Registra la instancia Singleton bajo el nombre dado. Mantener el registro simple, tendremos que almacenar una lista de objetos NameSingletonPair. Cada NameSingletonPair asigna un nombre a un singleton. La operación de búsqueda encuentra un singleton dado su nombre. Supondremos que una variable de entorno especifica el nombre del singleton deseado.

```
Singleton* Singleton::Instance () {
if (_instance == 0) {
const char* singletonName = getenv("SINGLETON");
// user or environment supplies this at startup

_instance = Lookup(singletonName);
// Lookup returns 0 if there's no such singleton
}
return _instance;
}
```

¿Dónde se registran las clases Singleton? Una posibilidad está en su constructor. Por ejemplo, una subclase MySingleton podría hacer lo siguiente:

```
MySingleton::MySingleton() {
// ...
Singleton::Register("MySingleton", this);
}
```

¡Por supuesto, el constructor no se llamará a menos que alguien instancia la clase, que hace eco al problema que el patrón Singleton está tratando de resolver! Podemos evitar este problema en C + + mediante la definición de una instancia estática de MySingleton. Por ejemplo, podemos definir static MySingleton theSingleton;

En el archivo que contiene la implementación de MySingleton. Ya no es la clase Singleton responsable de crear el singleton.
En cambio, su principal responsabilidad es hacer accesible el objeto singleton de elección en el sistema. El enfoque del objeto estático todavía tiene un posible inconveniente, es decir, que las instancias de deben crearse subclases, o bien no se registrarán.

Patrones relacionados: Muchos patrones se pueden implementar usando el patrón Singleton.

- Abstrac Factory
- Builder

- Prototype

Patrones Estructurales

Son los encargados de tratar la composición de las clases y/o objetos. Se refieren a como las clases y los objetos son organizados para conformar estructuras más complejas.

Adapter

Nombre del patrón: Adapter

Sinopsis:

Se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.

Contexto:

El patrón Adapter se puede aplicar cuando:

Queremos usar una clase existente, y ésta no tiene la interfaz que necesitamos.
Queremos crear una clase reutilizable que coopere con clases con las que no está relacionada. Por tanto, que no tendrán interfaces compatibles. (Sólo la versión de objetos) Necesitamos usar varias subclases existentes, pero sin tener que adaptar su interfaz creando una nueva subclase de cada una. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

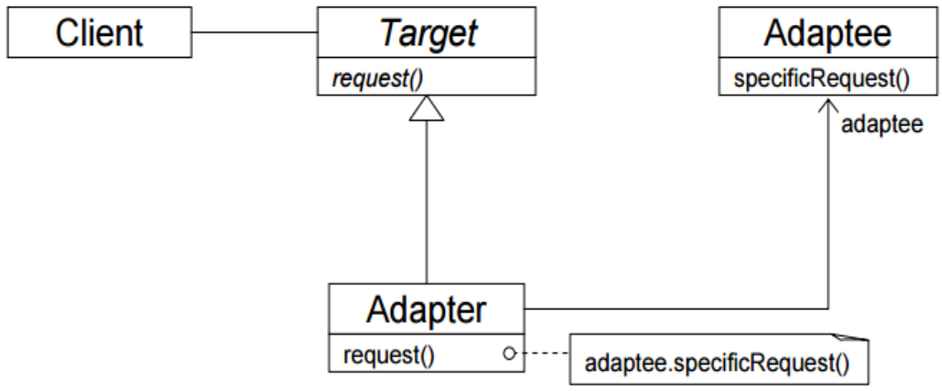
Causas:

Se quiere utilizar una clase que llame a un método a través de una interface, pero se busca utilizarlo con una clase que no implementa esa interface. Se busca determinar dinámicamente que métodos de otros objetos llama un objeto. No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.

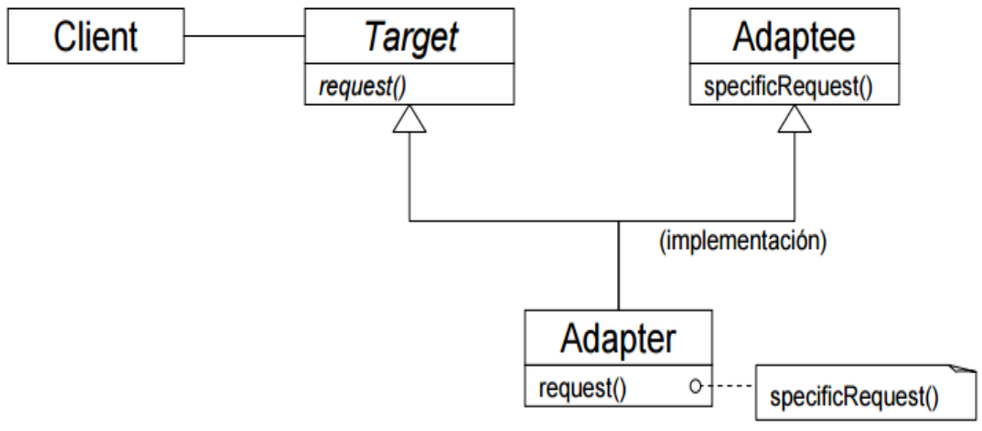
Solución:

Estructura:

Estructura – *object adapter*



Estructura – *class adapter*



Donde:

Target: define la interfaz específica de dominio que el cliente usa.
Client: colabora con los objetos que implementan la interfaz definida por el target.
Adaptee: define una interfaz existente que necesita adaptarse.
Aapter: adapta la interfaz del objeto adaptado a la definida por el target.

Estrategias:
Consecuencias:

- Object adapter
- Un adapter funciona con varios adaptees (el mismo adaptee y todas sus subclases)
- Dificulta sobrescribir el comportamiento del adaptee
- Class adapter
- El adapter hereda el comportamiento del adaptee, y puede sobrescribirlo
- No sirve para adaptar una clase y todas sus subclases
- Introduce un único objeto, no hace falta un nivel de indirección para obtener el adaptee.

Implementación:

Adaptador “conectable”
Maximiza la reutilización de las clases
Se adapta dinámicamente a una de varias clases
Java API:
Código:

```
public interface IPersonaNueva {

    public String getNombre() ;
    public void setNombre(String nombre) ;
    public int getEdad() ;
    public void setEdad(int edad);

}
```

```
public interface IPersonaVieja {

    public String getNombre();
    public void setNombre(String nombre);
    public String getApellido();
    public void setApellido(String apellido);
    public Date getFechaDeNacimiento();
    public void setFechaDeNacimiento(Date fechaDeNacimiento);

}
```

```
public class PersonaNueva implements IPersonaNueva {
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
```

```
public class PersonaVieja implements IPersonaVieja {
    private String nombre;
    private String apellido;
    private Date fechaDeNacimiento;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
```

```
public class ViejaToNuevaAdapter implements IPersonaNueva {
    private IPersonaVieja vieja;

    public ViejaToNuevaAdapter(IPersonaVieja vieja) {
        this.vieja = vieja;
    }

    public int getEdad() {
        GregorianCalendar c = new GregorianCalendar();
        GregorianCalendar c2 = new GregorianCalendar();
        c2.setTime(vieja.getFechaDeNacimiento());
        return c.get(1) - c2.get(1);
    }

    public String getNombre() {
        return vieja.getNombre() + " " + vieja.getApellido();
    }

    public void setEdad(int edad) {
        GregorianCalendar c = new GregorianCalendar();
        int anioActual = c.get(1);
        c.set(1, anioActual - edad);
        vieja.setFechaDeNacimiento(c.getTime());
    }

    public void setNombre(String nombreCompleto) {
        String[] name = nombreCompleto.split(" ");
        String firstName = name[0];
        String lastName = name[1];
        vieja.setNombre(firstName);
        vieja.setApellido(lastName);
    }

}
```

```
public static void main(String[] args) {

    PersonaVieja personaVieja = new PersonaVieja();
    personaVieja.setApellido("Perez");
    personaVieja.setNombre("Maxi");
    GregorianCalendar g = new GregorianCalendar();
    g.set(2000, 01, 01);
    // seteamos que nacio en el año 2000
    Date d = g.getTime();
    personaVieja.setFechaDeNacimiento(d);
    // hasta aqui creamos un PersonaVieja como se hacia antes

    // ahora veremos como funciona el adapter

    ViejaToNuevaAdapter personaNueva = new ViejaToNuevaAdapter(personaVieja);

    System.out.println(personaNueva.getEdad());
    System.out.println(personaNueva.getNombre());

    personaNueva.setEdad(10);
    personaNueva.setNombre("Juan Perez");

    System.out.println(personaNueva.getEdad());
    System.out.println(personaNueva.getNombre());

}
```

Bridge

Nombre del patrón: Bridge

Sinopsis:

El patrón Bridge, también conocido como Handle/Body, es una técnica usada en programación para desacoplar una abstracción de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra.

Desacoplar una abstracción de su implementación de modo que los dos puedan ser modificados de forma independiente.

Contexto:

Tenemos la necesidad de que la implementación de una abstracción sea modificada en tiempo de ejecución o nuestro sistema requiere que la funcionalidad (parcial o total) de nuestra abstracción esté desacoplada de la implementación para poder modificar tanto una como otra sin que ello obligue a la cambiar las demás clases

Causas:

- Queremos evitar enlaces permanentes entre una abstracción y una implementación.
- Tanto las abstracciones como las implementaciones deben ser extensibles por medio de subclases.
- Queremos que los cambios en la implementación de una abstracción no afecten al cliente.
- Necesitamos que la implementación de una característica sea compartida entre múltiples objetos.

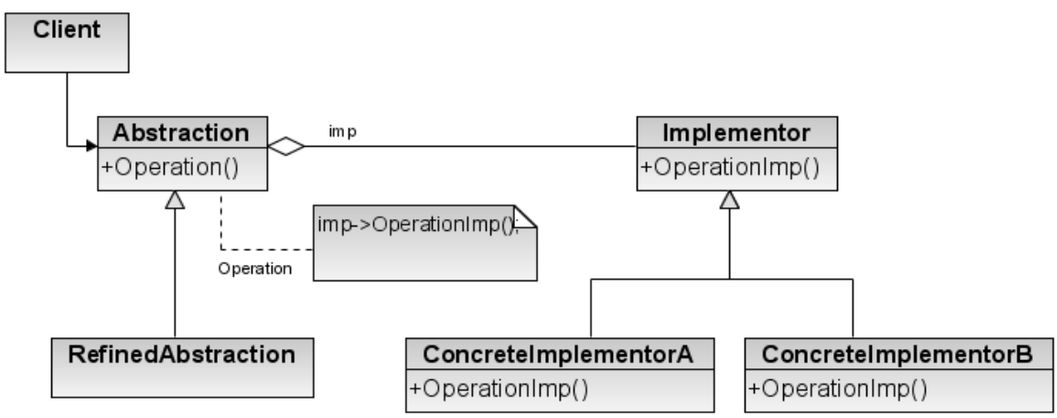
Solución:

Partimos de una abstracción base (clase abstracta o interfaz) que tendrá como atributo un objeto que será el que realice las funciones a implementar y que denominaremos implementador. Nuestra abstracción contendrá todas las operaciones que nuestro sistema requiera.

Por otro lado, tendremos el implementador, que será una interfaz que defina las operaciones necesarias para cubrir la funcionalidad que ofrece nuestra abstracción. Para dotar de funcionalidad a las operaciones definidas podremos crear diferentes implementadores concretos que implementen dicha interfaz.

Por último, debemos crear una clase que herede de nuestra abstracción para definir concretamente lo que hacen sus métodos, pero ésta deberá implementar la funcionalidad mediante el atributo que heredó del padre (que almacena un implementador).

Estructura:



Donde:

Client: Como siempre, se encargará de utilizar la abstracción de la que trata el problema.

Abstraction: Abstracción objetivo del problema. Contiene una referencia a un objeto que implemente la interfaz Implementator el cual servirá para definir la funcionalidad de la misma. Puede ser una clase abstracta o una interfaz.

Implementator: Interfaz que define las operaciones necesarias para implementar la funcionalidad de Abstraction.

RefinedAbstraction: Clase que hereda o implementa (según sea clase abstracta o interfaz respectivamente) de Abstraction. Extiende su funcionalidad basándose en el atributo que almacena al Implementator.

ConcreteImplementor: Implementación concreta de la funcionalidad definida por el Implementor. Puede haber varias implementaciones para la misma funcionalidad.

Consecuencias:

Positivas:

- Una implementación no se limita permanentemente a una interface.
- La implementación de una abstracción puede ser configurada y/o cambiada en tiempo de ejecución.
- Desacoplando Abstraction e Implementor también se eliminan las dependencias sobre la implementación en tiempo de compilación.
- Cambiar una implementación no requiere recompilar la clase Abstraction ni sus clientes.
- La capa de alto nivel de un sistema sólo tiene que conocer Abstraction e Implementor.
- Se pueden extender las jerarquías de Abstraction e Implementor sin que haya dependencias.
- Oculta los detalles de implementación a los clientes.

Negativas:

- Puede ser complicado de entender al principio.
- Añade complejidad.
- Problemática al no entender bien su funcionamiento.

Implementación:

Para ilustrar todo esto vamos a ver un ejemplo que representará la abstracción del envío de un paquete. Nuestra interfaz abstracta representará la agencia de transportes que realizará el envío:

```
public abstract class EmpresaMensajeria{

        protected IEnvio envio;


        protected EmpresaMensajeria(IEnvio envio){

                this.envio = envio;

        }


        public void recogerPaquete(){

                System.out.println('Se ha recogido el paquete.');

                envio.procesarEnvio();

        }


        public void enviarPaquete(){

                envio.enviar();

        }


        public void entregarPaquete(){

                envio.procesarEntrega();

                System.out.println('Se ha entregado el paquete.');

        }


        public void setEnvio(IEnvio envio){
```

A continuación, vamos a definir la interfaz IEnvio que representará al implementador del envío:

```
public interface IEnvio{

        public void procesarEnvio();

        public void enviar();

        public void procesarEntrega();

}
El siguiente paso es crear implementaciones de la interfaz IEnvio:
public class EnvioMar implements IEnvio{

        public void procesarEnvio(){

                System.out.println('El paquete se ha cargado en el barco.');

        }

        public void enviar(){

                System.out.println('El paquete va navegando por el mar.');

        }

        public void procesarEntrega(){

                System.out.println('El paquete se ha descargado en el puerto.');

        }
}
public class EnvioAire implements IEnvio{

        public void procesarEnvio(){

                System.out.println('El paquete se ha cargado en el avión.');

        }

        public void enviar(){

                System.out.println('El paquete va volando por el aire.');

        }

        public void procesarEntrega(){

                System.out.println('El paquete se ha descargado en el aeropuerto.');

        }

}
```

Ahora crearemos la empresa de transportes refinada:

```
public class EuroTransport extends EmpresaMensajeria{

        private String nif;

        public EuroTransport(String nif){

                IEnvio envioPorDefecto = new EnvioAire();

                super(envioPorDefecto);

                this.nif=nif;

        }


        public EuroTransport(String nif, IEnvio envio){

                super(envio);

                this.nif=nif;

        }


        public void identificarse(){

                System.out.println("Identificación: "+this.nif);

        }
}
```

Y por último hacemos que un cliente utilice nuestra abstracción:

```
public static void main(String[] args){

        // En primer lugar crearemos el objeto que representa a la emrpesa de mensa
jerio

        EmpresaMensajeria mensajero = new EuroTransport("0854752177");

        // Enviaremos un paquete vía aérea, que es la que esta empresa tiene pro de
fecto

        mensajero.recogerPaquete();

        mensajero.enviarPaquete();

        mensajero.entregarPaquete();

        // Ahora le decimos a la empresa que queremos enviar por mar

        mensajero.setEnvio(new EnvioMar());

        mensajero.recogerPaquete();

        mensajero.enviarPaquete();

        mensajero.entregarPaquete();
```

Patrones Relacionados: Adapter, Abstract Factory

COMPOSITE O COMPOSICION

Nombre: Composite o composicion

Sinopsis:

Nos indica cómo se organizan los objetos en memoria para obtener una composición recursiva o lo que es lo mismo una jerarquía en forma de árbol. Combina objetos en estructuras de árbol para representar jerarquías de parte todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Contexto:

Cuando desarrollar una aplicación como un editor de dibujos y sistemas de circuitos que permiten a los usuarios constituir diagramas complejos desde componentes simples.

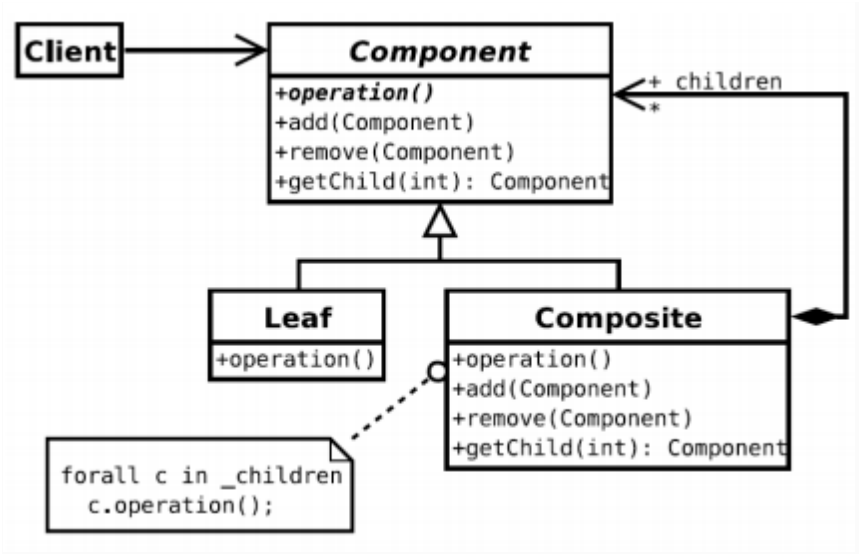
(Interfaces Graficas) Los objetos simples(hojas) y los contenedores de ellos están en diferentes maneras, aunque los clientes tratan ambos en la misma manera. La clave del patrón Composite: – Una clase abstracta que representa ambos
objetos los simples(hojas) y sus contenedores.

Causas:

- Quieres representar jerarquías de objetos todo-parte.
- Quieres ser capaz de ignorar la diferencia entre objetos individuales y composiciones de objetos. Los clientes tratarán a todos los objetos de la estructura compuesta uniformemente.

Solución:

Estructura: Estructura básica de la solución.



Participantes:

- **Component** (Grafico):
  - define la interfaz común para los objetos de la composición.
  - define la interfaz para acceder y gestionar los hijos.
  - implementa un comportamiento por defecto común a las subclases (opcional) define la interfaz para acceder al padre de un componente en la estructura recursiva, y la implementa si es apropiado
- **Leaf** (Linea, Rectangulo, Texto):
  - representa los objetos hoja (sin hijos) de la composición
  - define comportamiento para los objetos primitivos
- **Composite** (Dibujo):

- define comportamiento para los componentes que tienen hijos.
- almacena componentes hijo.
- implementa las operaciones de Component para la gestión de hijos.

- **Client**: o manipula los objetos de la composición a través de la interfaz Component.

Colaboraciones entre participantes:

- El cliente emplea la interfaz Componente (el API que actúa como superclase) para interactuar con los objetos de la composición:
  - Si se actúa sobre una hoja, entonces la petición es realizada directamente por la instancia correspondiente.
  - Si se actúa sobre una composición, en general se redirige la petición del componente a sus hijos y se realiza alguna acción adicional (posiblemente antes o después de redirigirle la petición).

Consecuencias:

- Define jerarquías de clases hechas de objetos primitivos y compuestos. Si el código cliente espera un objeto simple, puede recibir también uno compuesto.
- Simplifica el cliente. Objetos simples y compuestos se tratan homogéneamente.
- Facilita la incorporación de nuevos tipos de componentes.
- Puede hacer el diseño demasiado general. Es complicado restringir el tipo de componentes de un composite.

Implementación:

- Referencias explícitas a los padres
  - Simplifica algunas operaciones de la estructura compuesta
  - Definirlas en la clase Component
  - Gestionarlas al añadir/eliminar elementos de un Composite
- Compartir componentes
  - Útil para ahorrar memoria
  - La gestión de un componente con varios padres se complica
- Maximizar la interfaz del componente
  - Dar comportamiento por defecto que sobrescribirán las subclases
  - Ej: por defecto getChildren no devuelve hijos, lo cual es válido para las hojas, pero los compuestos deben sobrescribir la operación
- Declaración de las operaciones de gestión de hijos
  - Definirlas en la raíz Component y dar implementación por defecto
- Se obtiene transparencia, se pierde seguridad (¿cómo evitar que un cliente añada/elimine objetos a una hoja?)
  - Definirlas en Composite
- Se obtiene seguridad, se pierde transparencia (interfaz no uniforme)

- Si se pierde el tipo hay que hacer downcasting, lo cual es inseguro
- ¿Debe declarar Component una lista de componentes?
  - Penalización de espacio por cada hoja, incluso si no tiene hijos
- A veces los hijos tienen un orden
- ¿Quién debe borrar los componentes?
  - Si no hay recolector de basura, el Composite
  - Excepción si las hojas son inmutables y pueden estar compartidas

Veamos el Component:

```
public abstract class Component
{
    /// <summary>
    /// Variable donde su ámbito tiene mucha importancia en vista a la implementac
    /// </summary>
    protected string nombre;

    /// <summary>
    /// Constructor de la clase
    /// </summary>
    /// <param name="nombre">Nombre de la entidad a Crear</param>
    public Component(string nombre)
    {
        this.nombre = nombre;
    }

    /// <summary>
    /// Método Abstracto para agregar componentes, se utilizará en el composite.
    /// </summary>
    /// <param name="g"></param>
    public abstract void Agregar(Component g);
    public abstract void Eliminar(Component g);
    public abstract void Mostrar(int marca,System.Web.UI.Page p);

}
```

Como podemos observar importante es la declaración de la clase como abstracta, la variable "nombre" con su ámbito protegido y los tres métodos, dos para agregar y eliminar y una operación en este caso de "Mostrar".

Veamos El Composite:

```
public class Composite:Component
{
    private ArrayList elementos = new ArrayList();

    public Composite(string nombre)
        : base(nombre)
    {

    }
    public override void Agregar(Component g)
    {
        elementos.Add(g);
    }

    public override void Eliminar(Component g)
    {
        elementos.Remove(g);
    }

    public override void Mostrar(int marca,System.Web.UI.Page p)
    {

        p.Response.Write("<b>Marca</b>" + marca);

        foreach (Component g in elementos)
        {
            g.Mostrar(marca,p);
        }
    }
```

Para este caso vemos la declaración de un vector "ArrayList" donde voy a almacenar los components, se ven las implementaciones de cada uno de los métodos "Agregar", "Eliminar" y "Mostrar", presten atención a la palabra clave "override" que explica la reescritura del método abstracto, y el método mostrar que muestra un render recursivo sobre una página web de lo que tiene el composite.

Veamos el Leaf:

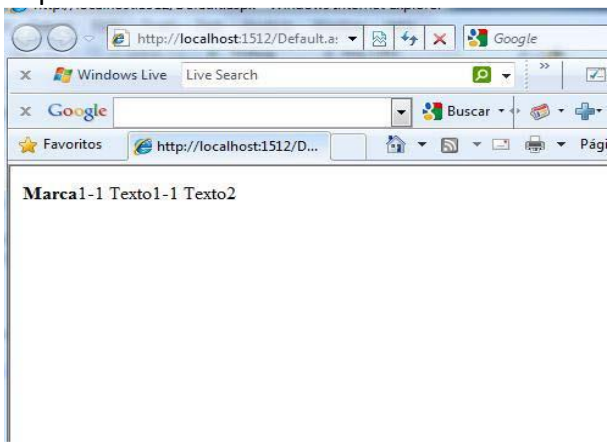
```
public class Leaf:Component
{
    public Leaf(string nombre)
        : base(nombre)
    {

    }
    public override void Agregar(Component g)
    {
        throw new NotImplementedException();
    }

    public override void Eliminar(Component g)
    {
        throw new NotImplementedException();
    }

    public override void Mostrar(int marca,System.Web.UI.Page p)
    {
        p.Response.Write("-" + marca + " " + nombre);
    }
}
```

En la figura anterior vemos que no están implementados los métodos "Agregar", "Eliminar" ya que no es de característica composite, sino un primitivo y si tiene detallado la operación en este caso "Mostrar".

Veamos el Cliente Web:

```
protected void Page_Load(object sender, EventArgs e)
{
    Composite ce = new Composite("Dibujo");
    ce.Agregar(new Leaf("Texto1"));
    ce.Agregar(new Leaf("Texto2"));

    ce.Mostrar(1, this);

}
```

Como podemos ver, la implementación es muy simple y la visualización recursiva también, veamos el resultado por pantalla:



Patrones relacionados:

- **Chain of resposability**: se utiliza en combinación con el patrón composite cuando hay que propagar los métodos hacia arriba, en el árbol desde las hojas hasta los nodos.
- **Flyweight**: permite compartir componentes u objetos primitivos en composite.
- **Iterator:** puede ser utilizado para encapsular el recorrido de los coposite (objetos complejos).
- **Visitor**: centraliza operaciones y comportamientos que de otra manera tendrían que ser divididos en las hojas y los composite.

DECORATOR

Nombre: Decorator

Sinopsis:

Responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

Contexto:

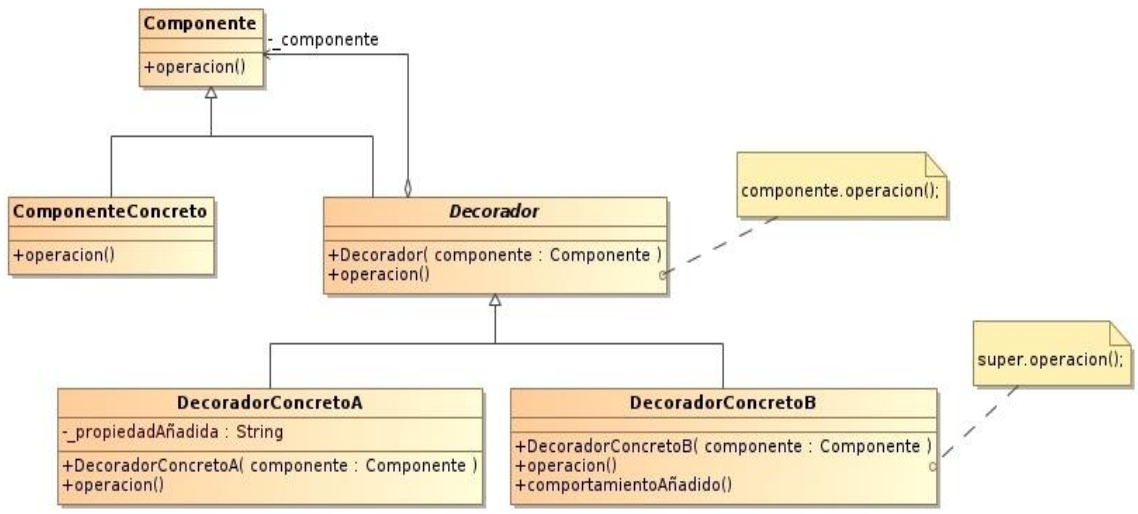
Es usado para añadir funcionalidades a una clase de forma dinámica, evitando las jerarquías de clases que se tienen construir en tiempo de compilación.

Solución:

Herencia: no es flexible, la funcionalidad se añade estáticamente.

Definir una clase "decoradora" que envuelve al componente, y le proporciona la funcionalidad adicional requerida: más flexible, transparente al cliente, se pueden anidar decoradores.

Estructura



Participantes:

- **Componente**: Define la interfaz para los objetos que pueden tener responsabilidades añadidas.

- **Componente Concreto**: Define un objeto al cual se le pueden agregar responsabilidades adicionales.

- **Decorador**: Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.

- **Decorador Concreto**: Añade responsabilidades al componente.

Colaboración entre participantes:

- El decorador redirige las peticiones al componente asociado.
- Opcionalmente puede realizar tareas adicionales antes y después de redirigir la petición.

Consecuencias:

- Más flexible que la herencia. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Además, evita la utilización de la herencia con muchas clases y también, en algunos casos, la herencia múltiple.
- Evita la aparición de clases con muchas responsabilidades en las clases superiores de la jerarquía. Este patrón nos permite ir incorporando de manera incremental responsabilidades.
- Genera gran cantidad de objetos pequeños. El uso de decoradores da como resultado sistemas formados por muchos objetos pequeños y parecidos.
- Puede haber problemas con la identidad de los objetos. Un decorador se comporta como un envoltorio transparente. Pero desde el punto de vista de la identidad de objetos, estos no son idénticos, por lo tanto, no deberíamos apoyarnos en la identidad cuando estamos usando decoradores.

Implementación:

Sistema de envío de mails corporativo

Component:

```java
public interface IEmail {
    public String getContents();
}
```

ConcreteComponent:

```java
public class Email implements IEmail {
    private String content;

    public Email(String content) {
        this.content = content;
    }

    @Override
    public String getContents() {
        return content;
    }
}
```

Decorator;

```java
public abstract class EmailDecorator implements IEmail
    IEmail originalEmail;
}
```

```java
public class ExternalEmailDecorator extends EmailDecorator {
    private String content;

    public ExternalEmailDecorator(IEmail basicEmail) {
        originalEmail = basicEmail;
    }

    @Override
    public String getContents() {
        content = addDisclaimer(originalEmail.getContents());
        return content;
    }

    private String addDisclaimer(String message) {
        return  message + "\n Disclaimer";
    }
}

public class SecureEmailDecorator extends EmailDecorator {
    private String content;

    public SecureEmailDecorator(IEmail basicEmail) {
        originalEmail = basicEmail;
    }

    @Override
    public String getContents() {
        content = encrypt(originalEmail.getContents());
        return content;
    }

    private String encrypt(String message) {
        return encryptedMessage;
    }
}
```

Main:

```
public class Sender() {
    public static void main(String[] args) {
        ...
    }

    public String generateEmail(IEmail email, int sentType) {
        String emailText = "";

        switch(sentType) {
            case EXTERNAL:
                EmailDecarator d1 = new ExternalEmailDecorator(email);
                emailText = d1.getContents();
                break;
            case SECURE:
                EmailDecarator d2 = new SecureEmailDecorator(email);
                emailText = d2.getContents();
                break;
            default:
                ...
        }

        return emailText;
    }
}
```

FACADE

Nombre: Facade

Sinopsis:

Proporcionar una interfaz unificada a un conjunto de interfaces en un subsistema. Define una interfaz de nivel superior que hace que el subsistema sea más fácil de usar.

Contexto:

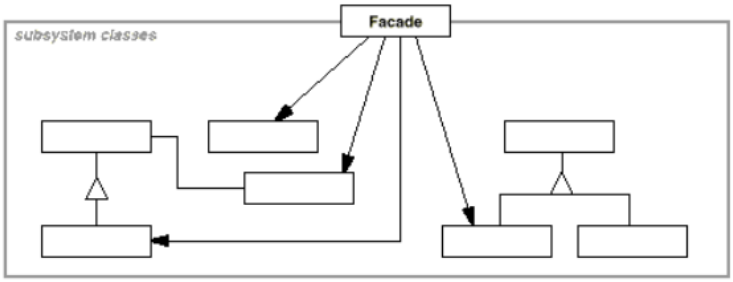
Simplificar el acceso a un conjunto de clases proporcionando una única clase que todos utilizan para comunicarse con dicho conjunto de clases.
Reducir la complejidad y minimizar dependencias.

Causas:

La estructuración de un sistema en subsistemas ayuda a reducir la complejidad. Un diseño común.
El objetivo es minimizar la comunicación y las dependencias entre subsistemas. Una Manera de lograr este objetivo es introducir un objeto de facade que proporciona una única Interfaz simplificada a las instalaciones más generales de un subsistema.

Solución:

Estructura:



Participantes: Facade(Compiler)
Sabe qué clases de subsistema son responsables de una solicitud.
Delega las solicitudes de cliente a objetos de subsistema apropiados.

Clases de subsistema (Scanner, Parser, ProgramNode, etc.)
Implementar la funcionalidad del subsistema.
Manejar el trabajo asignado por el objeto Facade.
No tienen conocimiento de la fachada; Es decir, no mantienen referencias

Colaboradores: Los clientes se comunican con el subsistema enviando solicitudes a Facade, que

- Los envía al objeto (s) del subsistema apropiado. Aunque el Objetos del subsistema realizar el trabajo real, la fachada puede tener que hacer el trabajo
- Para traducir su interfaz a interfaces de subsistema.
- Los clientes que utilizan la fachada no tienen que acceder a sus objetos del subsistema directamente.

Consecuencias:

1. Protege a los clientes de los componentes del subsistema, reduciendo así el número De los objetos que los clientes tratan y hacer el subsistema más fácil de usar.
2. Promueve el acoplamiento débil entre el subsistema y sus clientes. A menudo el

Componentes de un subsistema están fuertemente acoplados. El acoplamiento débil le permite variar los componentes del subsistema sin afectar a sus clientes. Las fachadas ayudan a nivelar un sistema y las dependencias entre objetos. Pueden eliminar Complejas o circulares. Esto puede ser una consecuencia.

El cliente y el subsistema se implementan independientemente.
Reducir las dependencias de la compilación es vital en sistemas de software grandes. Tú
Quieren ahorrar tiempo minimizando la recompilación cuando cambian las clases del subsistema.
La reducción de las dependencias de compilación con fachadas puede limitar la recompilación.
Necesaria para un pequeño cambio en un subsistema importante. Una fachada también puede:

Simplificar los sistemas de portado a otras plataformas, ya que es menos
La construcción de un subsistema requiere la construcción de todos los demás.
3. No impide que las aplicaciones utilicen clases de subsistema si necesitan
a. Por lo tanto, puede elegir entre facilidad de uso y generalidad.

Implementación:

A considerar:

1. Reducción del acoplamiento cliente-subsistema. El acoplamiento entre los clientes y
Se puede reducir aún más al hacer de Facade una clase abstracta
Con subclases concretas para diferentes implementaciones de un subsistema. Entonces
Los clientes pueden comunicarse con el subsistema a través de la interfaz de
Resumen Clase de fachada. Este acoplamiento abstracto evita que los clientes sepan
Aplicación de un subsistema.
Una alternativa a la subclase es configurar un objeto Facade con
Objetos del subsistema. Para personalizar la fachada, simplemente reemplace uno o más de
Sus objetos del subsistema.
2. Clases públicas y privadas de subsistemas. Un subsistema es análogo a una clase
En que ambos tienen interfaces, y ambos encapsulan algo: una clase
Encapsula estado y operaciones, mientras que un subsistema encapsula clases.
Y así como es útil pensar en la interfaz pública y privada de
Una clase, podemos pensar en la interfaz pública y privada de un subsistema.
La interfaz pública a un subsistema consiste en clases que todos los clientes
puede acceder; La interfaz privada es sólo para extensores de subsistema. los
La clase de fachada es parte de la interfaz pública, por supuesto, pero no es la
sólo parte. Otras clases de subsistemas suelen ser públicas. Por ejemplo,
Las clases Parser y Scanner en el subsistema de compilación forman parte del

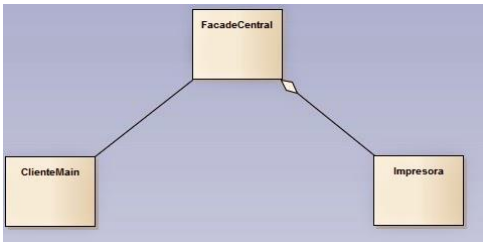
Pública.
Hacer que las clases de subsistema sean privadas sería útil, pero pocos orientados a objetos Los idiomas lo apoyan. Tradicionalmente, C ++ y Smalltalk han tenido un espacio de nombres global para las clases. Recientemente, sin embargo, la estandarización C ++ Comité agregó espacios de nombre al lenguaje [Str94], que le permitirá
Exponer sólo las clases de subsistema público.

Código:

Problema:

Un cliente necesita conectarse a una impresora, pero desconoce el nombre del impreso y la configuración de la misma solo sabe el texto que la envía a la misma que se debe imprimir, el cliente (como todo cliente/usuario) al meter manos en este dispositivo puede ocasionar problemas graves.

Solución:



Bueno partiendo de la base, tratemos de identificar los componentes, por un lado, tenemos el cliente, por el otro la impresora, pero si al cliente le damos el control de la impresora corremos el riesgo que meta la pata, la solución a esto es crear una interfase la cual se conecte con la impresora y que la misma la administre y a su esta sea consumida por el cliente.

Impresora:

```
package ar.com.patronesdisenio.facade.impresora;
/**
* @author nconde
*/
public class Impresora {

private String nombre;
private String tipoHoja;
private String texto;

public String getNombre() {
return nombre;
}
```

```
public void setNombre(String nombre) {

this.nombre = nombre;

}

public String getTipoHoja() {

return tipoHoja;

}
public void setTipoHoja(String tipoHoja) {
this.tipoHoja = tipoHoja;
}
public String getTexto() {
return texto;
}
public void setTexto(String texto) {
this.texto = texto;
}
}
```

FacadeCentral:

```
package ar.com.patronesdisenio.facade.central;

import ar.com.patronesdisenio.facade.impresora.Impresora;

/**
*
* @author nconde
*
*/
public class FacadeCentral {

private Impresora impresora;

public void imprimir(String texto){
impresora = new  Impresora();
impresora.setNombre("NICO-Printer");
impresora.setTipoHoja("A4");
impresora.setTexto(texto);
}

}
```

ClienteMain:

```
package ar.com.patronesdisenio.facade.cliente;
import ar.com.patronesdisenio.facade.central.FacadeCentral;

public class ClienteMain {

/**
* @param args
*/
public static void main(String[] args) {

FacadeCentral facadeCentral  = new FacadeCentral();
facadeCentral.imprimir("Texto a imprimir");
}

}
```

En este ejemplo vemos como el cliente desconoce como se imprime el nombre de la impresora y la existencia de la misma solo ve lo que deseamos el "imprimir".

Patrones relacionados:

Abstract Factory (99) se puede utilizar con Facade para proporcionar una interfaz para crear subsistema de una manera independiente del subsistema. Abstract Factory también puede ser utilizado como una alternativa a la fachada para ocultar las clases específicas de la plataforma.

Mediador (305) es similar a Facade en que abstrae la funcionalidad de las clases Sin embargo, el propósito de Mediator es abstraer la comunicación arbitraria entre objetos de colega, a menudo centralizando la funcionalidad que no pertenece en cualquiera de ellos. Los colegas de un mediador son conscientes y se comunican con

El mediador en lugar de comunicarse directamente entre sí. En contraste, una fachada simplemente abstrae la interfaz a los objetos del subsistema para facilitarlo usar. No define una nueva funcionalidad, y las clases de subsistema no saben acerca de ello.

FLYWEIGHT

Patrón: FlyWeight

Sinopsis:

Compartir una parte común del estado de un objeto para hacer más eficiente la gestión de un número
elevado de objetos.

Contexto:

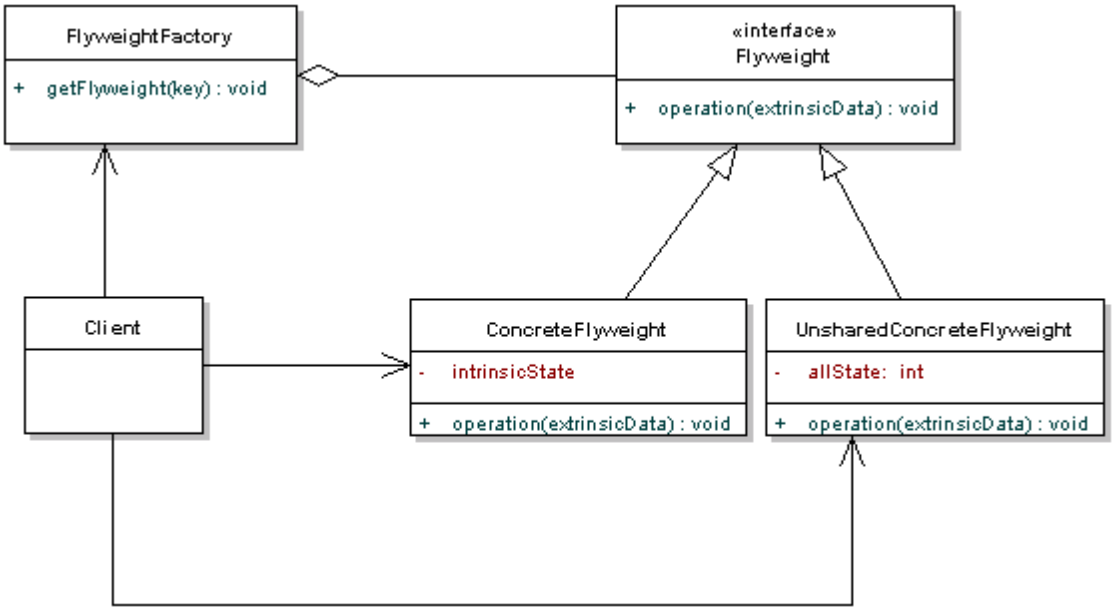
Algunas aplicaciones pueden usar objetos durante todo su diseño, pero esto puede ocasionar implementaciones costosas. La mayoría de editores de documentos tienen formato de texto y facilidades de edición que son implementados por alguna extensión. Típicamente editores de documento orientados a objetos usan objetos para representar elementos embebidos como tablas o figuras, así también utilizan objetos para representar cada carácter, manejar el editor de esta manera ofrece flexibilidad al sistema, pues pueden ser dibujados y formateados uniforme-mente, con figuras y tablas, pero la desventaja es que un documento de texto puede tener miles de caracteres, por eso tener un objeto por carácter implica un gran costo debido a la memoria que puede consumir. Flyweight permite compartir objetos ligeros, para hacer el programa más liviano. Los objetos pueden compartir estados intrínsecos que no dependen del contexto, pero no pueden compartir los estados extrínsecos que dependen del contexto.

Causas:

Se necesita eliminar o reducir la redundancia cuando se tiene gran cantidad de objetos que comparten bastante información.
Nuestro soporte tiene memoria limitada y esta necesita ser aprovechada óptimamente.
La identidad propia de los objetos es irrelevante.

Solución:

Estructura:



Participantes:

Client: Trabaja con una referencia a un Flyweight. Establece los estados extrínsecos de los objetos.
FlyweightFactory: Crea y maneja objetos flyweight asegurándose que se compartan adecuadamente.
Cuando el cliente solicita un flyweight, FlyweightFactory proporciona una instancia existente y si no existe la crea.
Flyweight: Interfaz que contiene métodos para el establecimiento, acceso y modificación de las propiedades extrínsecas. Deberá ser implementada por las instancias del Flyweight.
ConcreteFlyweight: Implementa la interfaz Flyweight y añade el almacenamiento de las características intrínsecas (si las hay) y deben poder ser compartidos.
UnsharedConcreteFlyweight: Todos los Flyweight no tienen por qué ser compartidos, la implementación de la interfaz habilita la compartición, pero no la fuerza. Es común que los objetos UnsharedConcreteFlyweight tengan hijos ConcreteFlyweight en algún punto de la jerarquía.

Consecuencias:

Positivas:

Simplifica el uso de sistemas complejos con tareas redundantes.
Oculta al cliente la complejidad real del sistema.
Reduce el acoplamiento entre el subsistema y los clientes.

Negativas:

Aumenta la complejidad de los objetos.
Aumenta el número de clases del sistema.

Implementación:

IArbolLigero.cs (Flyweight Interface)

IArbolLigero.cs*

Flyweight Pattern

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Flyweight_Pattern
{
    public interface IArbolLigero
    {
        String GetTipo();
        void Dibujar(long x,long y);
    }
}
```

Arbol.cs (ConcreteFlyweight)

Arbol.cs*

Flyweight Pattern | Flyweight_Pattern.Arbol | tipo

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Flyweight_Pattern
{
    public class Arbol : IArbolLigero
    {

        private String tipo;

        public Arbol(String tipo)
        {
            this.tipo = tipo;
        }

        public void Dibujar(long x, long y)
        {
            Console.WriteLine("Árbol ["+this.GetTipo()+"] dibujado en las coordenadas ("+x+","+y+")");
        }

        public String GetTipo()
        {
            return this.tipo;
        }
    }
}
```

FabricaDeArboles.cs (FlyweightFactory)

FabricaDeArboles.cs*

Flyweight Pattern | Flyweight_Pattern.FabricaDeArboles | getArbol(string tipo)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Flyweight_Pattern
{
    class FabricaDeArboles
    {
        private static Dictionary<String, Arbol> arboles = new Dictionary<string, Arbol>();

        public FabricaDeArboles()
        {

        }

        public IArbolLigero getArbol(String tipo)
        {


            if(!arboles.ContainsKey(tipo))
            {
                arboles.Add(tipo,new Arbol(tipo));
                Console.WriteLine("Agregando árbol ["+tipo+"] al pool");
            }

            return arboles[tipo];
        }
    }
}
```

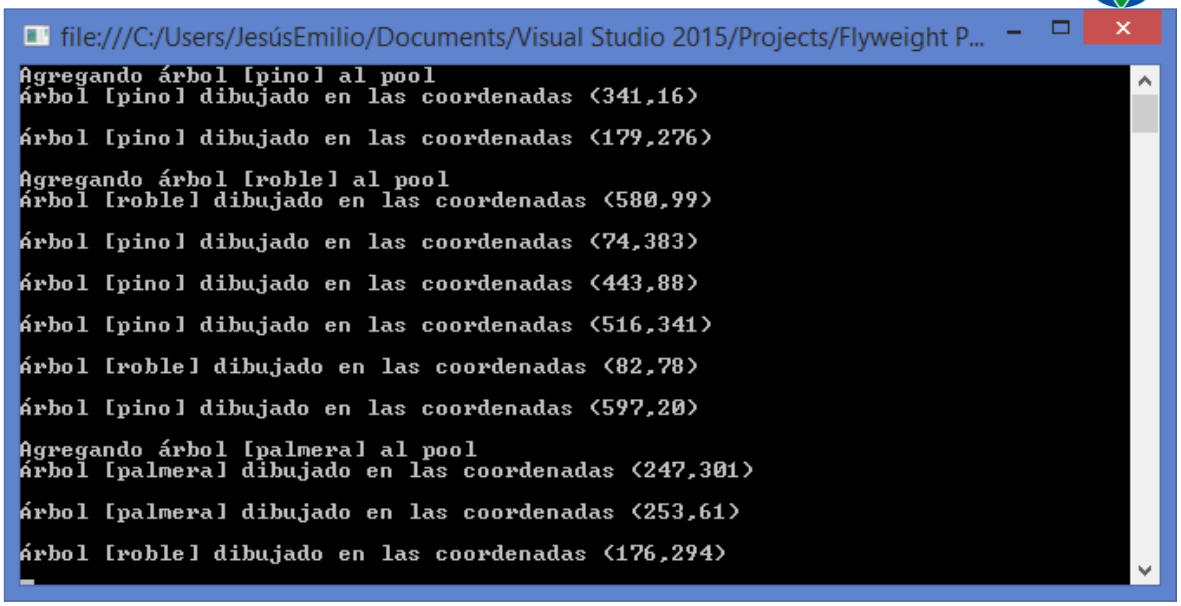
Programa.cs (Client)

Programa.cs

Flyweight Pattern | Flyweight_Pattern.Programa | Main(string[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Flyweight_Pattern
{
    class Programa
    {
        static void Main(string[] args)
        {
            FabricaDeArboles fabricaDeArboles = new FabricaDeArboles();

            String[] tipoArboles = { "pino", "palmera", "roble" };

            Random random = new Random();
            for (int i = 0; i <= 10000; i++)
            {
                fabricaDeArboles.getArbol(tipoArboles[random.Next(tipoArboles.Length)]).Dibujar(random.Next(600), random.Next(400));
                Console.ReadLine();
            }

        }
    }
}
```

Resultado:



Patrones relacionados:
Abstract Factory, Composite, State y Strategy.

PROXY

Patrón: Proxy

Sinopsis:

Este patrón consiste en interponer un intermediario (Proxy) entre un objeto y los demás que lo utilizan.

Contexto:
El patrón proxy se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero. Dependiendo de la función que se desea realizar con dicha referencia podemos distinguir diferentes tipos de proxies:

proxy remoto: representante local de un objeto remoto.
proxy virtual: crea objetos costosos bajo demanda.
proxy de protección: controla el acceso al objeto original

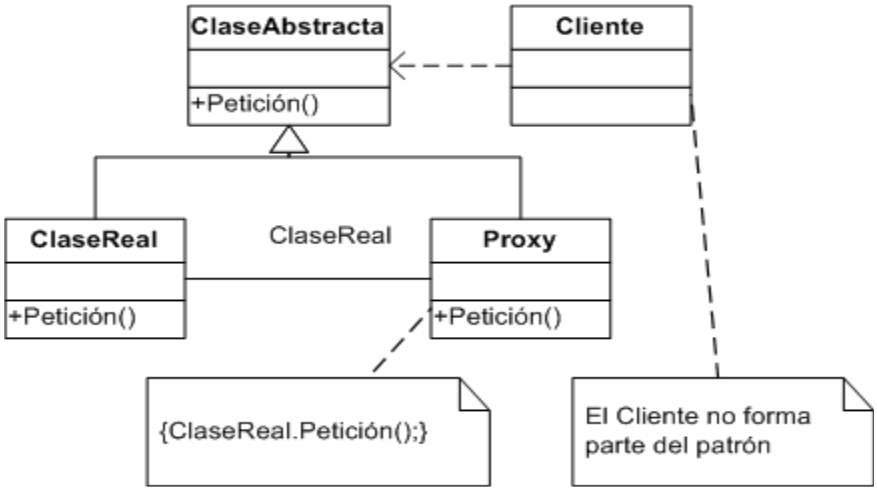
proxy de referencia inteligente: sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto (ej. contar número de referencias al objeto real, cargar un objeto persistente bajo demanda en memoria, control de concurrencia de acceso tal como bloquear el objeto para impedir acceso concurrente, …).

Causas:

Se suele utilizar para implementar comportamientos "vagos" (lazy). Por ejemplo, si tenemos muchos
objetos imagen en un documento, se tardaría mucho tiempo en abrir el documento al cargar las
imágenes de disco. Para evitarlo podemos sustituir los objetos imagen por objetos proxyImagen, con el
mismo interfaz, pero que solamente cargan la imagen cuando se va a visualizar.

Solución:

Estructura:



Consecuencias:

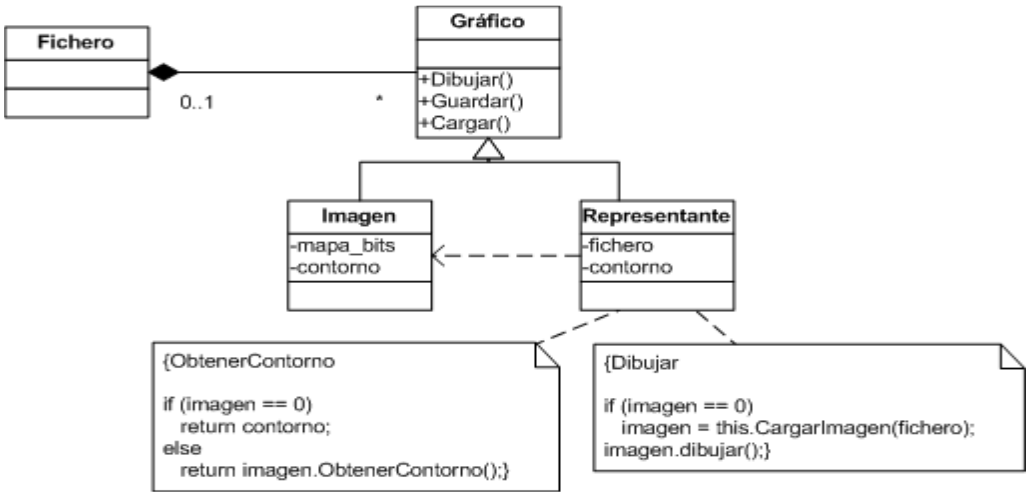
El uso de un proxy introduce un nivel de indirección adicional con diferentes usos:

Un proxy remoto oculta el hecho de que un objeto reside en otro espacio de direcciones.
Un proxy virtual puede realizar optimizaciones, como la creación de objetos bajo demanda.
El proxy de protección y las referencias inteligentes permiten realizar diversas tareas de mantenimiento adicionales al acceder a un objeto.

Además, su uso también permite realizar una optimización COW (copy-on-write), puesto que copiar un objeto grande puede ser costoso, y si la copia no se modifica, no es necesario incurrir en dicho gasto. Además, el sujeto mantiene un número de referencias, y sólo

cuando se realiza una operación que modifica el objeto, éste se copia. Es útil por tanto para retrasar la replicación de un objeto hasta que cambia.

Implementación:



Patrones relacionados: Adapter, decorador

Conclusión

Los patrones creacionales ayudan a resolver los problemas relacionados con la creación de instancias de una clase, y así separar la implementación del cliente de la de los objetos que se utilizan. Nos ayudan a abstraer y encapsular su creación.

Singleton: Asegura que una determinada clase sea instanciada una y sólo una vez, proporcionando un único punto de acceso global a ella.
Abstract Factory: Provee una interfaz para crear familias de objetos relacionados o que dependen entre si, sin especificar sus clases concretas.
Builder: Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones. Simplifica la construcción de objetos con estructura interna compleja y permite la construcción de objetos paso a paso.
Factory Method: Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
Prototype: Facilita la creación dinámica de objetos mediante la definición de clases cuyos objetos pueden crear duplicados de si mismos. Estos objetos son llamados prototipos.

Los patrones estructurales son utilizados para crear clases u objetos que incluidos dentro de estructuras más complejas.

Decorator: Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
Adapter: Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permiten que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
Bridge: Desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

Composite: Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Facade: Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema se más fácil de usar.

Flyweight: Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

Proxy: Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Decorator: Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Los patrones de diseño describen la solución a problemas que se repiten una y otra vez en nuestros sistemas, de forma que se puede usar esa solución siempre que haga falta. Capturan el conocimiento que tienen los expertos a la hora de diseñar. Ayudan a generar software “maleable” (software que soporta y facilita el cambio, la reutilización y la mejora). Son guías de diseño, no reglas rigurosas.

Referencias

Design Patterns: Elements of Resusable Object Oriented Software.
Patrones de diseño: elementos de software orientado a objetos reusables pág. 144-152, 208 217
https://msdn.microsoft.com/es-es/library/bb972272.aspx#EDAA
http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf
http://es.slideshare.net/ikercanarias/patrones-de-diseo-de-software-14836338
http://siul02.si.ehu.es/~alfredo/iso/06Patrones.pdf
https://www.genbetadev.com/paradigmas-de-programacion/diseno-con-patrones-y-fachadas

http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf

http://java-white-box.blogspot.mx/2014/10/patrones-de-diseno-patron-facade.html

https://es.wikipedia.org/wiki/Decorator_(patr%C3%B3n_de_dise%C3%B1o)

https://www.genbetadev.com/metodologias-de-programacion/patrones-de-diseno-decorator

https://jesusramirezguerrero.com/2014/08/26/patrones-de-diseno-en-java/

http://codejavu.blogspot.mx/2013/07/ejemplo-patron-de-diseno-decorator.html

https://danielggarcia.wordpress.com/2014/03/10/patrones-estructurales-iii-patron-decorator/

http://informaticapc.com/patrones-de-diseno/decorator.php

http://programacion.net/articulo/patrones_de_diseno_v_patrones_de_creacion_prototipo_1005

http://tratandodeentenderlo.blogspot.mx/2010/02/patrones-de-diseno-prototype.html

http://migranitodejava.blogspot.mx/2011/05/builder.html

https://danielggarcia.wordpress.com/2014/02/19/patrones-de-creacion-ii-patron-builder-constructor/

http://programacion.net/articulo/patrones_de_diseno_iii_patrones_de_creacion_builder_1002

https://danielggarcia.wordpress.com/2014/03/17/patrones-estructurales-iv-patron-bridge/

https://es.wikipedia.org/wiki/Bridge_(patr%C3%B3n_de_dise%C3%B1o)

http://informaticapc.com/patrones-de-diseno/bridge.php

http://programacion.net/articulo/patrones_de_diseno_viii_patrones_estructurales_bridge_1010

http://migranitodejava.blogspot.mx/2011/05/factory-method.html

https://www.ecured.cu/Factory_Method

http://jjmorra.blogspot.mx/2014/07/factory-method-o-metodo-de-fabricacion.html

http://arantxa.ii.uam.es/~eguerra/docencia/0708/05%20Composite.pdf

https://nbortolotti.blogspot.mx/2009/05/disenando-con-patrones-una-vista-al.html