JUAN SEBASTIAN JARAMILLO SILVA - 30000056254

DISEÑO DE SOFTWARE CREACIÓN

ABSTRACT FACTORY

¿Qué es?

El patrón Abstract Factory nos permite crear, mediante una interfaz, conjuntos o familias de objetos (denominados productos) que dependen mutuamuente y todo esto sin especificar cuál es el objeto concreto.

Cuando usar:

- Para apoyar familias de objetos relacionados o dependientes.
- Encapsular las dependencias de la plataforma para hacer que una aplicación sea portátil.
- Para evitar que el código del cliente use el operador 'nuevo'.
- Para intercambiar fácilmente la plataforma subyacente con cambios mínimos.

Intención

Proporcionar una interfaz para crear familias de objetos relacionados o dependientes.

sin especificar sus clases concretas.

Componentes

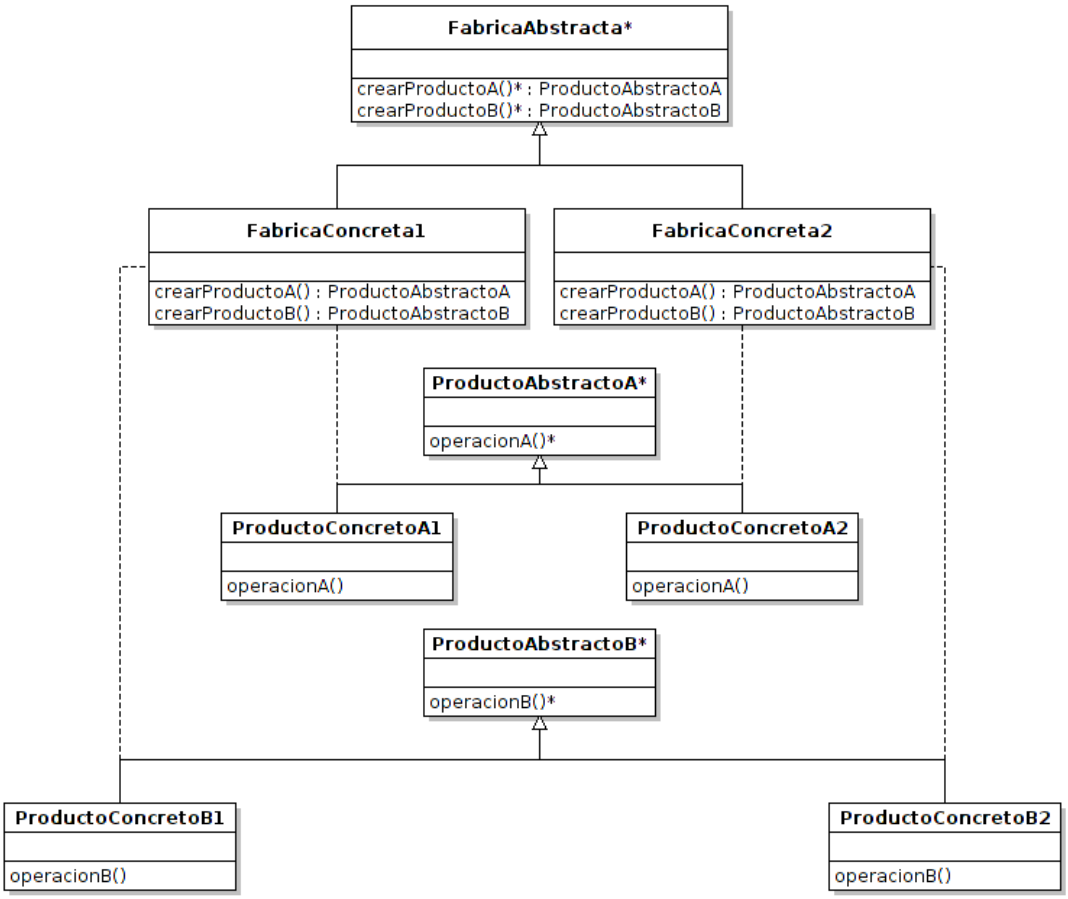
1. Una clase de Fábrica abstracta (pública)

2. Implementaciones de fábrica para varias familias (protegidas)

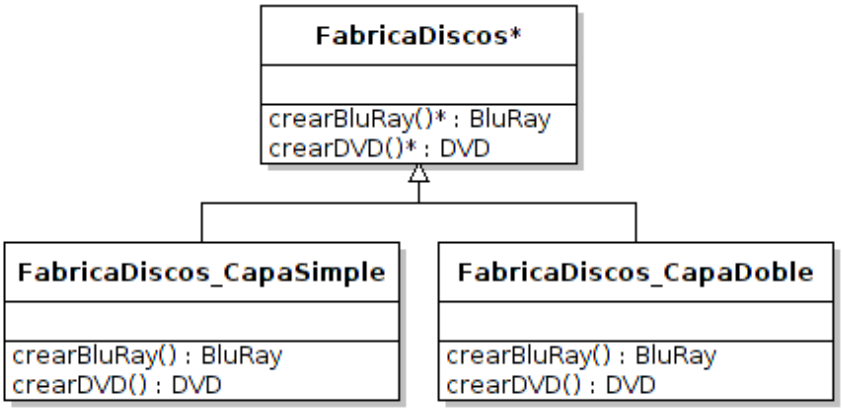
3. Interfaces para diversos productos (públicos)

4. conjunto de implementaciones de productos para varias familias (protegido)

Estructura:



Implementación



La factoría abstracta la definiríamos mediante un interface:

```
public interface FabricaDiscos {
    public BluRay crearBluRay();
     public DVD crearDVD();
}
```

Y sobre ese interface implementamos una de las fábricas concretas, en este caso la de FabricaDiscos_CapaSimple:

```
public class FabricaDiscos_CapaSimple implements FabricaDiscos {
    @Override
    public BluRay crearBluRay() {
        return new BluRay_CapaSimple();
    }
    @Override
    public DVD crearDVD() {
        return new DVD_CapaSimple();
    }
}
```

Veamos como quedaría el código fuente en Java para utilizar el patrón Abstract Factory:

```
FabricaDiscos fabrica;
DVD dvd;
BluRay bluray;
fabrica = new FabricaDiscos_CapaSimple();
dvd = fabrica.crearDVD();
bluray = fabrica.crearBluRay();
System.out.println(dvd);
System.out.println(bluray);
fabrica = new FabricaDiscos_CapaDoble();
```

```
dvd = fabrica.crearDVD();

bluray = fabrica.crearBluRay();

System.out.println(dvd);

System.out.println(bluray);
```

En el código vemos que sobre la fábrica podemos crear objetos de diferentes tipos y que podríamos ir creciendo en productos atendiendo a nuestras necesidades.

```
fabrica = new FabricaDiscos_CapaSimple();

dvd = fabrica.crearDVD();

bluray = fabrica.crearBluRay();
```

FACTORY METHOD

Cuándo usar

- Para imponer la codificación de la interfaz en lugar de la implementación
- Transferir la responsabilidad de instanciación de la clase de cliente al método de fábrica
- Para desacoplar la implementación del programa cliente

Intención

Defina una interfaz para crear un objeto, pero deje que las subclases decidan qué

clase para instanciar. El Método de Fábrica permite que una clase difiera la instanciación a

subclases

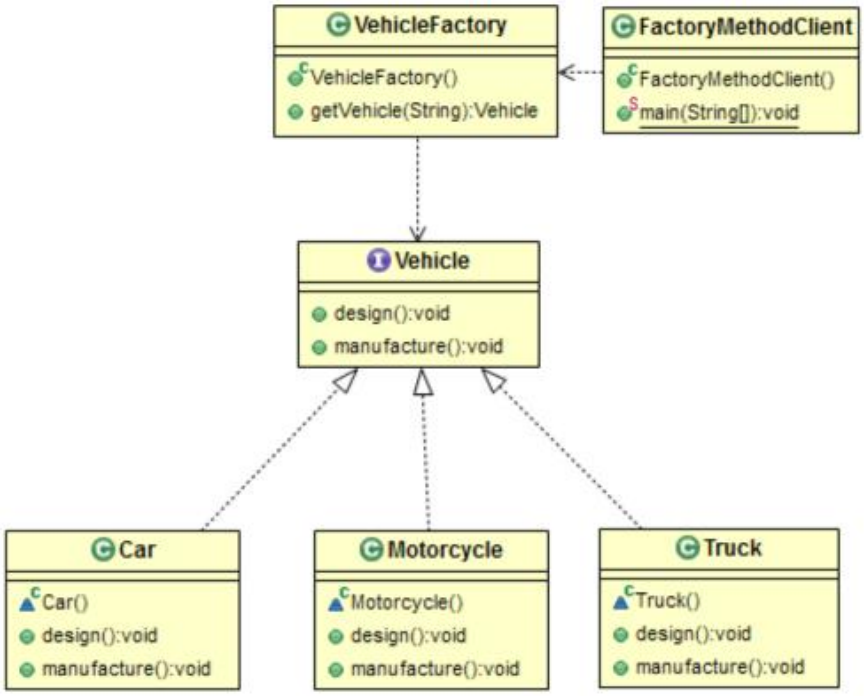
Componentes

1. Una interfaz (o) clase abstracta (pública)

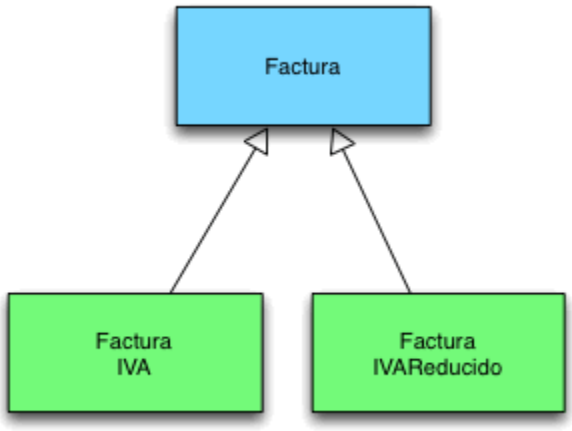
2. Conjunto de subclases de implementación (privado)

3. Un método de fábrica (público)

Estructura



Implementación



El código será el siguiente:

```
package com.arquitecturajava;
public abstract class Factura {
private int id;
 private double importe;
 public int getId() {
 return id;
 }
 public void setId(int id) {
 this.id = id;
 }
 public double getImporte() {
 return importe;
 }
 public void setImporte(double importe) {
 this.importe = importe;
 }
 public abstract double getImporteIva();
}
package com.arquitecturajava;
public class FacturaIva extends Factura{
@Override
 public double getImporteIva() {
 // TODO Auto-generated method stub
 return getImporte()*1.21;
 }
}
package com.arquitecturajava;
public class FacturaIvaReducido extends Factura{
```

```
@Override
 public double getImporteIva() {
 // TODO Auto-generated method stub
 return getImporte()*1.07;
 }
}
```

Como vemos la clase Factura es una clase abstracta de la cual heredan nuestras dos clases concretas que implementan el cálculo del IVA. Vamos a construir una Factoría para que se encargue de construir ambos objetos de la jerarquía.

```
package com.arquitecturajava;
public class FactoriaFacturas {
public static Factura getFactura(String tipo) {
 if (tipo.equals("iva")) {
 return new FacturaIva();
 }
 else {
 return new FacturaIvaReducido();
 }
 }
```

Si nos fijamos la clase lo único que hace es instanciar un objeto u otro dependiendo del tipo que le solicitemos. Eso en un principio parece poco práctico. Pero vamos a ver como queda el programa main:

```
package com.arquitecturajava;
public class Principal {
 public static void main(String[] args) {
 Factura f= FactoriaFacturas.getFactura("iva");
 f.setId(1);
 f.setImporte(100);
 System.out.println(f.getImporteIva());
 }
}
```

BUILDER

Cuándo usar

- Para evitar tratar con objetos inconsistentes cuando el objeto necesita ser creado en varios pasos.

- Para evitar demasiados argumentos de constructor.

- Para construir un objeto que debería ser inmutable.

- Encapsular la lógica de creación completa.

Intención

Separe la construcción de un objeto complejo de su representación para que el mismo proceso de construcción puede crear diferentes representaciones.

Componentes

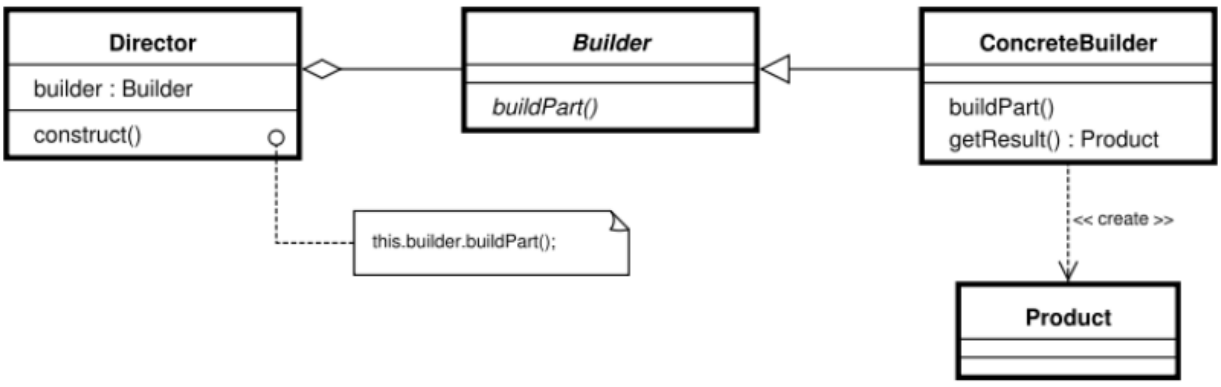
1. La clase Builder especifica una interfaz abstracta para crear partes de un Objeto del producto.

2. El ConcreteBuilder construye y reúne partes del producto. implementando la interfaz del generador. Define y realiza un seguimiento de la representación que crea y proporciona una interfaz para guardar el producto.

3. La clase Director construye el objeto complejo usando el Generador interfaz.

4. El Producto representa el objeto complejo que se está construyendo.

Estructura



Ejemplo:

Vamos a repasar los conceptos mediante un caso práctico en Java. Siguiendo la línea del artículo anterior, crearemos distintos tipos de coches (berlina, coupé y monovolúmen) abstrayéndonos de su representación.

En primer lugar definiremos el producto objeto. Fijarse en la manera de definir los setter, ésto es una buena práctica a la hora de asignar múltiples propiedades a la vez, ya que podremos hacerlo de la forma: coche.cilindrada(2000).potencia(300)...

```
public class Coche
{
    private int cilindrada = 0;
    private int potencia = 0;
    private String tipo = “”;
    private int num_asientos = 0.
    public Coche cilindrada(int cilindrada)
    {
        this.cilindrada = cilindrada;
        return this;
    }
    public Coche potencia(int potencia)
    {
        this.potencia = potencia;
        return this;
    }
    public Coche tipo(String tipo)
    {
        this.tipo = tipo;
        return this;
    }
```

```
    public Coche numAsientos(int num_asientos)
    {
        this.num_asientos = num_asientos;
        return this;
    }
}
```

A continuación, crearemos un constructor abstracto capaz de construir todos los tipos de coche:

```
abstract class ConstructorCoches
{
    protected Coche coche;
    public void nuevo(){coche = new Coche();}
    public Coche obtenerCoche(){return coche;}
    public abstract void construirMotor();
    public abstract void construirCarroceria();
}
```

El siguiente paso es definirlos Constructores concretos:

```
public class ConstructorCochesAudi extends ConstructorCoches
{
    public void construirMotor()
    {
        cohe.cilindrada(2995).potencia(300);
    }
    public void construirCarroceria()
    {
        cohe.tipo("Audi A7 Sportback 3.0 TFSI quattro S tronic 7 vel.").num_asientos (5);
    }
}
public class ConstructorCochesBMW extends ConstructorCoches
```

```
{
    public void construirMotor()
    {
        cohe.cilindrada(4395).potencia(560);
    }
    public void construirCarroceria()
    {
        cohe.tipo("BMW Serie 5 2016").num_asientos (5);
    }
}
public class ConstructorCochesTesla extends ConstructorCoches
{
    public void construirMotor()
    {
        cohe.potencia(560);
    }
    public void construirCarroceria()
    {
        cohe.tipo("Tesla Model S").num_asientos (5);
    }
}
```

Ahora toca especificar al Director, que será el que actuará de intermediario entre el usuario de los Constructores y los propios Constructores:

```
public class Concesionario
{
    private ConstructorCoches constructorCoches;
    public void establecerConstructor(ConstructorCoches cc){this.constructorCoches=cc;}
    public void obtenerCoche(){return constructorCoches.obtenerCoche();}
     public void construirCoche()
```

```
    {
        constructorCoches.nuevo();
        constructorCoches.construirMotor();
        constructorCoches.construirCarroceria();
    }
}
```

Y para finalizar, un cliente que use el constructor:

```
public static void main(String[] args)
{
    Concesionario director = new Concesionario();
    ConstructorCoches constructor;
    if(args[0].equals("Audi"))
    {
        constructor = new ConstructorCochesAudi();
    }
    else if(args[0].equals("BMW"))
    {
        constructor = new ConstructorCochesBMW();
    }
    else if(args[0].equals("Tesla"))
    {
        constructor = new ConstructorCochesTesla();
    }
    director.establecerConstructor(constructor);
    director.construirCoche();
    Coche producto = director.obtenerCoche();
}
```

SINGLETON

Cuando usar

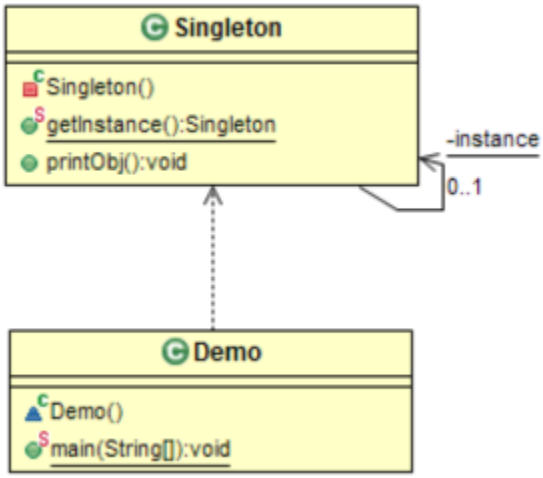
- La aplicación necesita" solo una instancia "de una clase.
- Para tener un control completo sobre la creación de la instancia.
-

Intención
Asegúrese de que una clase tenga solo una instancia y proporcione un punto de acceso global a lt.

Componentes

1. Clase Singleton

Estructura



Implementación

En el siguiente código en Java mostramos un ejemplo de uso de este patrón de diseño, en el que se intenta obtener cinco veces una instancia de la clase Coche:

Main.java:

```
package Singleton;

public class Main

{

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

```
    {
        for(int num=1; num<=5; num++)
        {
            Coche.getInstancia();
        }
    }
}
```

Coche.java:

```
package Singleton;
public class Coche
{
     private static Coche instancia;
    private Coche() {
    }
     public static Coche getInstancia()
    {
         if (instancia == null) {
             instancia = new Coche();
            System.out.println("El objeto ha sido creado");
        }
        else {
            System.out.println("Ya existe el objeto");
        }
         return instancia;
    }
}
```

PROTOTYPE

Cuando usar

- Para mejorar el rendimiento cuando la creación de objetos es costosa y el tiempo consumidor.
- Para simplificar y optimizar la creación de múltiples objetos que tendrán principalmente los mismos datos

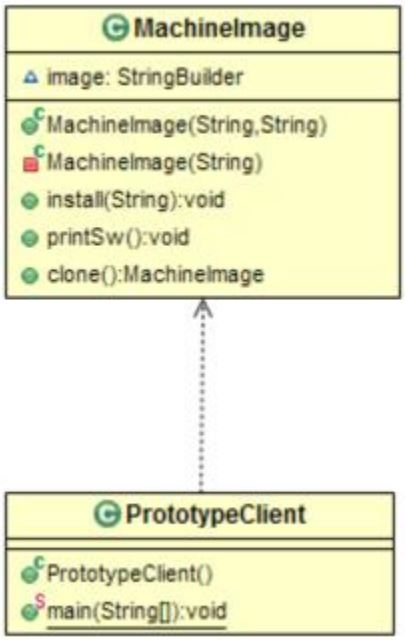
Intención

Especifique los tipos de objetos para crear utilizando una instancia prototípica, y crear nuevos objetos copiando este prototipo.

Componentes

1. Una clase que implementa la interfaz clonable (pública)

Estructura



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);
            }
        }
}
```

DEPENDENCY INJECTION

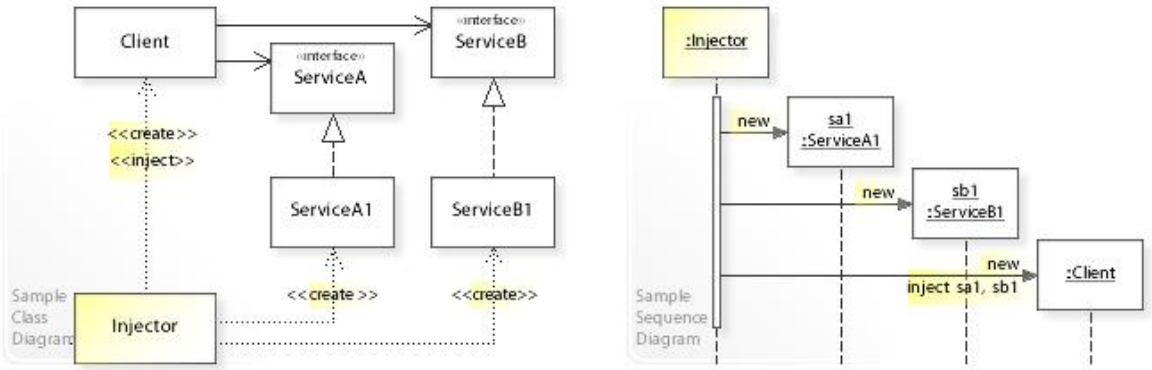
¿Qué es?

Es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos. Esos objetos cumplen contratos que necesitan nuestras clases para poder funcionar (de ahí el concepto de dependencia). Nuestras clases no crean los objetos que necesitan, sino que se los suministra otra clase 'contenedora' que inyectará la implementación deseada a nuestro contrato.

Formas de inyectar las dependencias

En java, y en general en los distintos lenguajes hay distintas formas de inyectar dependencias.2Esto se puede lograr de forma manual o mediante frameworks como Spring, para lograr esta versatilidad, la inyección de dependencias se apoya en la programación orientada a interfaces.

Estructura



Implementación

En el constructor

Usando el constructor del objeto. Por ejemplo si una clase A tiene la dependencia de un objeto con el interfaz B

```
public class A {
 private B dependency;
 public A(B instancedepency)
 {
  this.dependency=instancedepency;
 }
 //... En su implementación A usa el objeto dependency
}
```

En un método

Usando un método, típicamente el método suelen ser un setter.

```
public class A {
 private B dependency;
 public setDependecy(B instancedepency)
 {
  this.dependency=instancedepency;
 }
 //... En su implementación A usa el objeto dependency
```

}

En una variable de instancia

Usando una variable de instancia o propiedad.

```
public class A {
 public B dependency;
 //... En su implementación A usa el objeto dependency
}
```

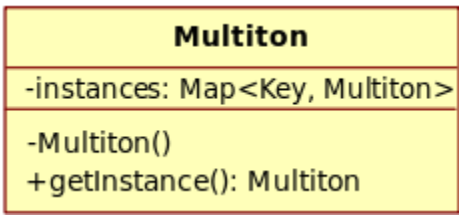
MULTITON

¿Qué es?

el patrón de Multiton es un patrón de diseño similar al Singleton , que permite sólo una instancia de una clase que se creará. El patrón Multiton amplía el concepto Singleton para gestionar un mapa de las instancias con nombre como pares de valores clave.

En lugar de tener una única instancia por aplicación (por ejemplo, el java.lang.Runtimeobjeto en el lenguaje de programación Java) el patrón Multiton en vez asegura una única instancia por clave .

Estructura



Implementación

En Java, el patrón Multiton se puede implementar utilizando un tipo enumerado , con los valores del tipo que corresponde a las instancias. En el caso de un tipo enumerado con un único valor, esto da el patrón singleton.

En C #, también podemos utilizar enumeraciones, como muestra el siguiente ejemplo:

```
using System.Collections.Generic;
public enum MultitonType {
   ZERO,
   ONE,
   TWO
```

```
};
public class Multiton {
   private static readonly Dictionary<MultitonType, Multiton> instances =
      new Dictionary<MultitonType, Multiton>();
   private int number;
   private Multiton(int number) {
      this.number = number;
   }
   public static Multiton GetInstance(MultitonType type) {
      // lazy init (not thread safe as written)
      // Recommend using Double Check Locking if needing thread safety
      if (!instances.ContainsKey(type)) {
         instances.Add(type, new Multiton((int)type));
      }
      return instances[type];
   }
   public override string ToString() {
      return "My number is " + number.ToString();
   }
   // Sample usage
   public static void Main(string[] args) {
      Multiton m0 = Multiton.GetInstance(MultitonType.ZERO);
      Multiton m1 = Multiton.GetInstance(MultitonType.ONE);
      Multiton m2 = Multiton.GetInstance(MultitonType.TWO);
      System.Console.WriteLine(m0);
      System.Console.WriteLine(m1);
      System.Console.WriteLine(m2);
   }
}
```

DISEÑO DE SOFTWARE – ESTRUCTURA

ADAPTER

Cuando usar

- Para envolver una clase existente con una nueva interfaz,
- Para realizar la correspondencia de impedancia

Intención

Convierta la interfaz de una clase en otra interfaz que los clientes esperan.

El adaptador permite que las clases trabajen juntas que de otra manera no podrían debido a interfaces incompatibles

Componentes

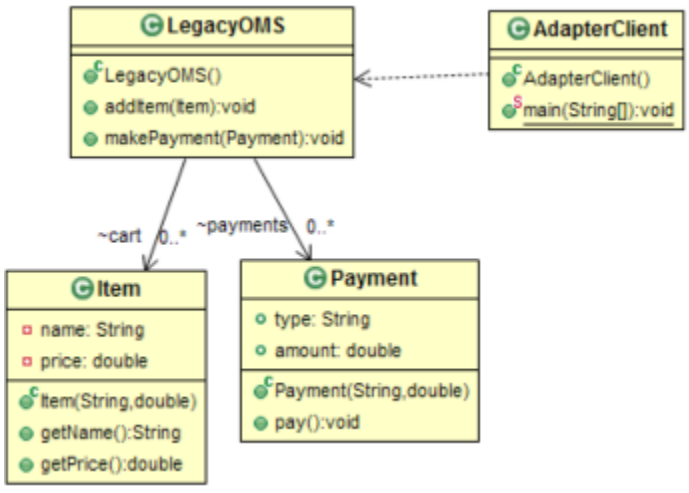
1. Destino: define la interfaz específica del dominio que utiliza el Cliente.

2. Adaptador: adapta la interfaz Adaptado a la interfaz de destino.

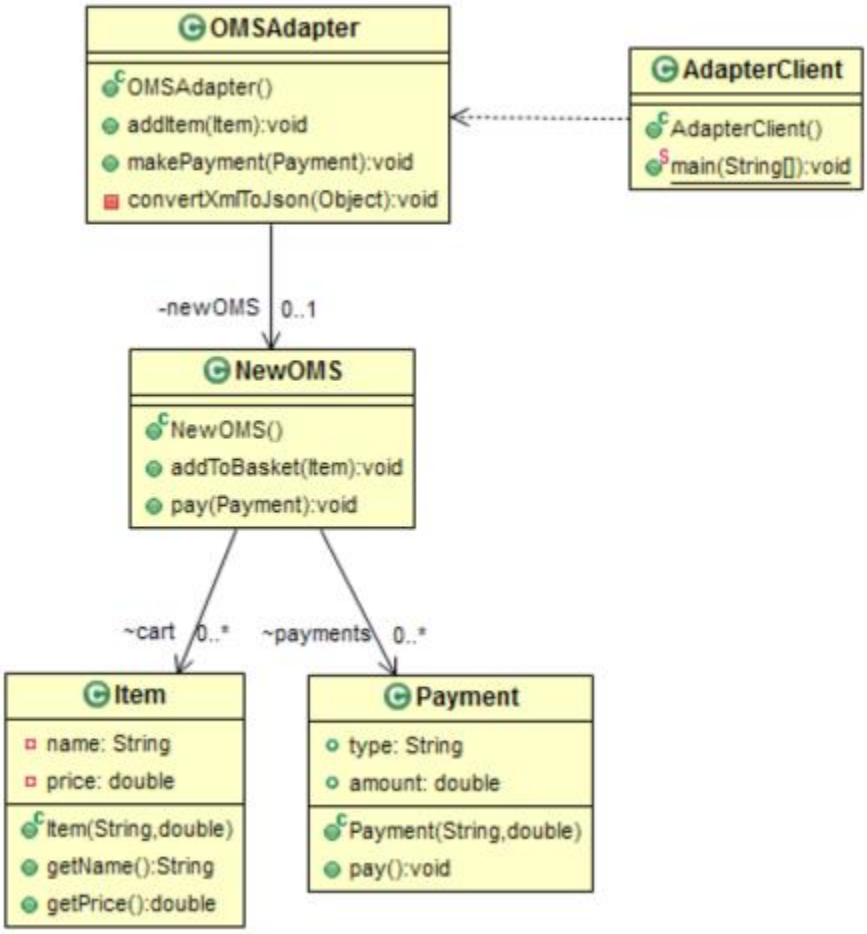
3. Adaptado: define una interfaz existente que necesita adaptación.

4, Cliente: colabora con objetos que se ajustan a la interfaz de destino.

Estructura – Antes



Estructura - Después



Implementación

Clase Motor.

Esta clase es desde la cual heredaran los diferentes tipos de motores, provee los métodos comunes (encender, acelerar, apagar) para el funcionamiento de los mismos.

```
public abstract class Motor {

  abstract public void encender();

  abstract public void acelerar();

  abstract public void apagar();

}
```

Motores Común y Económico.

Esta clase representa la estructura de los motores normales con los que el sistema funciona, básicamente heredan de la clase Motor y realizan el funcionamiento básico que esta provee.

```
public class MotorComun extends Motor {
    public MotorComun(){
        super();
        System.out.println("Creando el motor comun");
    }
    @Override
    public void encender() {
        System.out.println("encendiendo motor comun");
    }
    @Override
    public void acelerar() {
        System.out.println("acelerando el motor comun");
    }
    @Override
    public void apagar() {
        System.out.println("Apagando motor comun");
    }
}
```

Clase MotorElectricoAdapter.

Aquí se establece el puente por medio del cual la clase incompatible puede ser utilizada, hereda de la clase Motor y por medio de la implementación dada, realiza la comunicación con la clase a adaptar usando para esto una instancia de la misma...

```
public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;
    public MotorElectricoAdapter(){
        super();
```

```
        this.motorElectrico = new MotorElectrico();
        System.out.println("Creando motor Electrico adapter");
    }
    @Override
    public void encender() {
        System.out.println("Encendiendo motorElectricoAdapter");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }
    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico...");
        this.motorElectrico.moverMasRapido();
    }
    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

Clase MotorElectrico.

Esta es la clase adaptable, como vemos a pesar de ser un motor posee características muy diferentes a los demás tipos de motores del sistema, por lo tanto no puede heredar directamente de la clase Motor, en vez de esto, es accedida por la clase Adapter...

```
public class MotorElectrico {
    private boolean conectado = false;
    public MotorElectrico() {
        System.out.println("Creando motor electrico");
```

```
    this.conectado = false;
}
public void conectar() {
    System.out.println("Conectando motor electrico");
    this.conectado = true;
}
public void activar() {
    if (!this.conectado) {
        System.out.println("No se puede activar porque no " +
          "esta conectado el motor electrico");
    } else {
        System.out.println("Esta conectado, activando motor" +
          " electrico....");
    }
}
public void moverMasRapido() {
    if (!this.conectado) {
        System.out.println("No se puede mover rapido el motor " +
          "electrico porque no esta conectado...");
    } else {
        System.out.println("Moviendo mas rapido...aumentando voltaje");
    }
}
public void detener() {
    if (!this.conectado) {
        System.out.println("No se puede detener motor electrico" +
          " porque no esta conectado");
    } else {
        System.out.println("Deteniendo motor electrico");
```

```
        }
    }
    public void desconectar() {
        System.out.println("Desconectando motor electrico...");
        this.conectado = false;
    }
```

Clase Aplicación.

Finalmente esta clase representa el Cliente del sistema que usa los diferentes tipos de motores, como vemos desde aquí se hacen los llamados sin importar cual es la lógica detrás de estos, por medio del patrón Adapter llamamos a los mismos métodos encender(), acelerar() o apagar().

```
private void usarMotorComun() {
    Motor motor = new MotorEconomico();
    motor = new MotorComun();
    motor.encender();
    motor.acelerar();
    motor.apagar();
}
private void usarMotorElectrico() {
    Motor motor = new MotorElectricoAdapter() ;
    motor.encender();
    motor.acelerar();
    motor.apagar();
}
private void usarMotorEconomico() {
    Motor motor = new MotorEconomico();
    motor.encender();
    motor.acelerar();
    motor.apagar();
}
```

BRIDGE

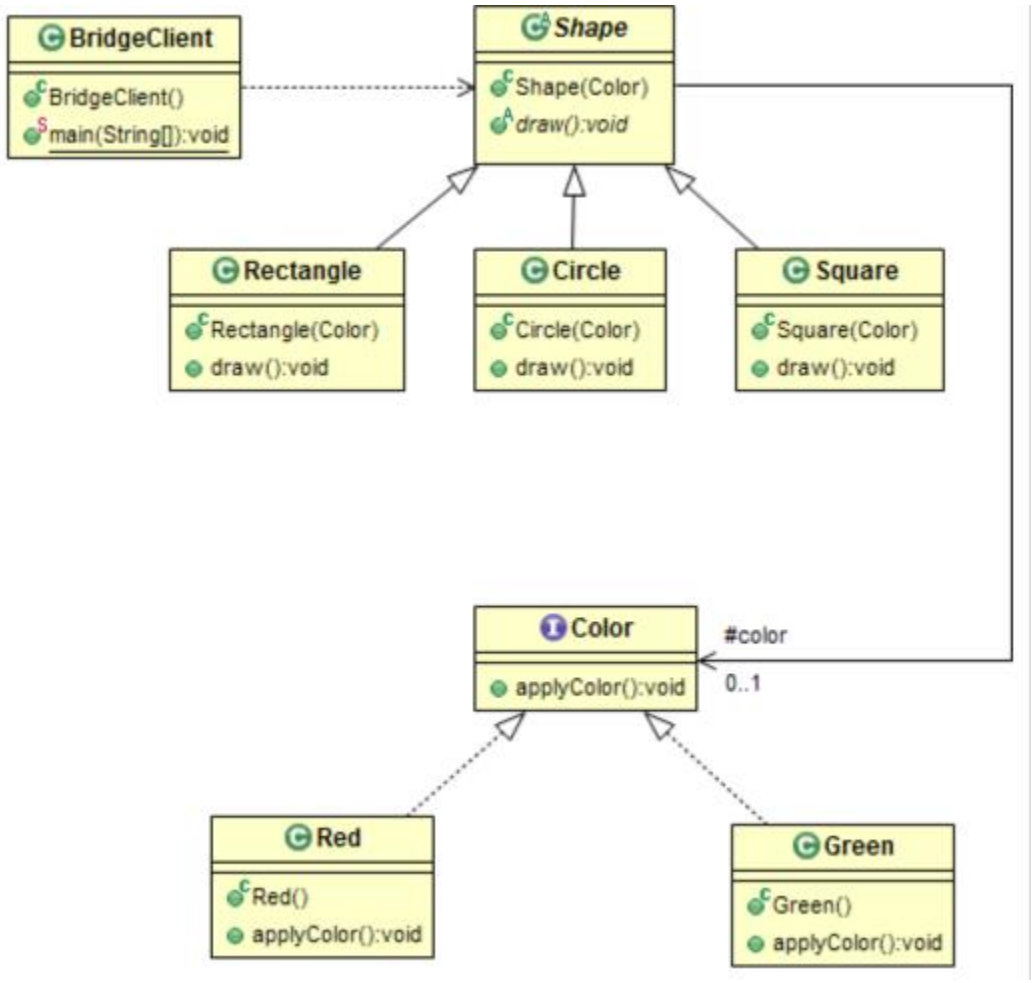
Cuándo usar

- Cuando se requiere el enlace en tiempo de ejecución de la implementación.
- Para admitir una proliferación de clases resultante de una interfaz acoplada y numerosas implementaciones,
- Para compartir una implementación entre múltiples objetos y para mapear jerarquías de clases ortogonales.

Intención

Desacoplar una abstracción de su implementación para que los dos puedan variar independientemente.

Estructura



Implementación

```
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){
        this.envio=envio;
    }
 public void getEnvio(){
        return this.envio;
    }
}
```

A continucación vamos a definir la interfaz IEnvio que representará al implemntador 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 mensajerio
    EmpresaMensajeria mensajero = new EuroTransport("0854752177");
    // Enviaremos un paquete vía aérea, que es la que esta empresa tiene pro defecto
    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();
}
```

COMPOSITE

Cuando usar

- Tener una colección jerárquica de entidades primitivas y compuestas.
- Para crear una estructura de manera que los objetos en la estructura puedan ser tratado de la misma manera.

Intención

Componga objetos en estructuras de árbol para representar jerarquías de parte completa.

Compuesto permite a los clientes tratar objetos individuales y composiciones de objetos uniformemente

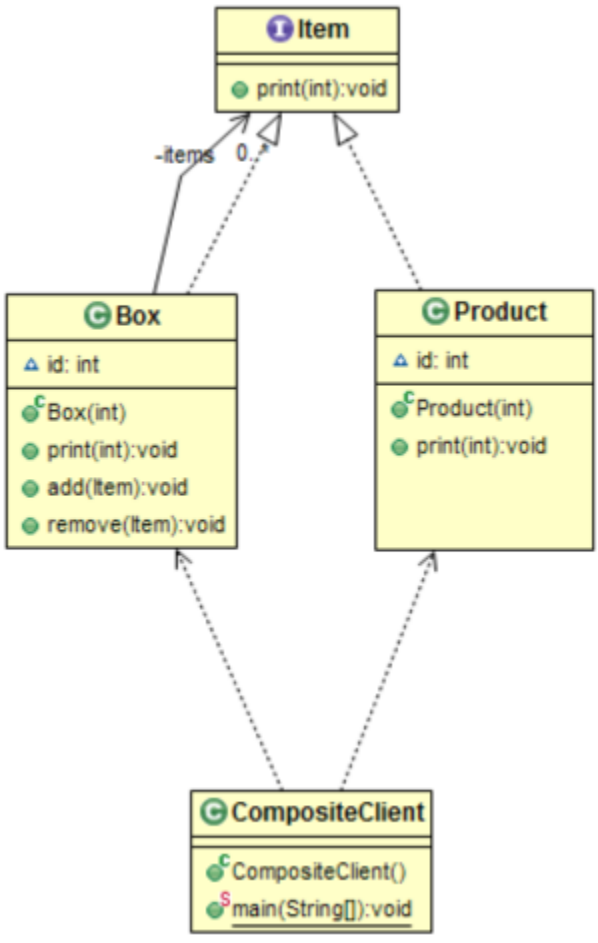
Componentes

1. Una interfaz para todos los objetos de la composición.

2. Un elemento de hoja que es el bloque de construcción de la composición.

3. Un elemento compuesto que puede contener elementos de hoja y / o composición

Estructura



Implementación

Ejemplo:

Vamos a ver un ejemplo para dejar todo más claro. Tenemos que diseñar un sistema que permita dibujar planos de una zona concreta. Las zonas pueden ser una ciudad, un barrio, una calle, una avenida, una plaza o una travesía. En primer lugar definiremos una interfaz que represente la zona a dibujar, que hará de Component:

```
interface Zona {
    public void generarPlano();
    public String obtenerTipo();
    public void agregarZona(Zona z);
    public Zona subzona(int i);
}
```

El siguiente paso es definir las zonas que pueden estar compuestas por otras zonas. Por ejemplo, una ciudad tiene varios barrios que a su vez están compuestos por domicilios. Lo haremos mediante la implementación de la interfaz zona:

```
public class Ciudad implements Zona{
    private String nombre;
    private ArrayList subzonas;
    public Ciudad(String nombre){
        this.nombre = nombre;
        subzonas = new ArrayList();

    public void generarPlano(){
        System.out.println("CIUDAD: "+this.nombre);
        for(int i=0; i<subzonas.size();i++){
            subzonas.get(i).generarPlano();
        }
    }
    public void obtenerTipo(){
        return "ciudad";}
```

```
    public void agregarZona(Zona z){
        // Las ciudades se subdividen en barrios
        if(z.getTipo().equals("barrio")){
            subzonas.add(z);
        } else {
            System.out.println("No se puede añadir "+z.getTipo()+" directamente a una ciudad.");
        }
    }
    public Zona subzona(int i){
        return subzonas.get(i);
    }
}
public class Barrio implements Zona{
    private String nombre;
    private ArrayList subzonas;
    public Barrio(String nombre){
        this.nombre = nombre;
        subzonas = new ArrayList();
    }
    public void generarPlano(){
        System.out.println("BARRIO: "+this.nombre);
        for(int i=0; i<subzonas.size();i++){
            subzonas.get(i).generarPlano();
        }
    }
    public void obtenerTipo(){
        return "barrio";
    }
```

```
public void agregarZona(Zona z){
        // Un barrio no puede tener ni barrios ni ciudades dentro de el.
        if(!(z.getTipo().equals("ciudad") ||
            (z.getTipo().equals("barrio")){
            subzonas.add(z);
        } else {
            System.out.println("No se puede añadir "+z.getTipo()+" directamente a un barrio.");
        }
    }
    public Zona subzona(int i){
        return subzonas.get(i);
        }
    }
```

Nos queda por implementar el resto de zonas, las que no son composiciones (zonas finales u hojas) o son composiciones sin hijos:

```
public class Calle implements Zona{
    private String nombre;
    public Calle(String nombre){
        this.nombre = nombre;
    }
    public void generarPlano(){
        System.out.println("CALLE: "+this.nombre);
    }
    public void obtenerTipo(){
        return "calle";
    }
    public void agregarZona(Zona z){
        System.out.println("No se pueden agregar zonas a "+z.getTipo());
    }
```

```
public Zona subzona(int i){
        System.out.println("Este elemento no contiene subzonas");
    }
}
public class Avenida implements Zona{
    private String nombre;
    public Avenida(String nombre){
        this.nombre = nombre;
    }
    public void generarPlano(){
        System.out.println("AVENIDA: "+this.nombre);
    }
    public void obtenerTipo(){
        return "avenida";
    }
    public void agregarZona(Zona z){
        System.out.println("No se pueden agregar zonas a "+z.getTipo());
    }
    public Zona subzona(int i){
        System.out.println("Este elemento no contiene subzonas");
    }
}
public class Plaza implements Zona{
    private String nombre;
    public Plaza(String nombre){
        this.nombre = nombre;
    }
    public void generarPlano(){
        System.out.println("PLAZA: "+this.nombre);
```

```
    }
    public void obtenerTipo(){
        return "plaza";
    }
    public void agregarZona(Zona z){
        System.out.println("No se pueden agregar zonas a "+z.getTipo());
    }
    public Zona subzona(int i){
        System.out.println("Este elemento no contiene subzonas");
    }
}
public class Travesia implements Zona{
    private String nombre;
    public Travesia(String nombre){
        this.nombre = nombre;
    }
    public void generarPlano(){
        System.out.println("TRAVESIA: "+this.nombre);
    }
    public void obtenerTipo(){
        return "travesia";
    }
    public void agregarZona(Zona z){
        System.out.println("No se pueden agregar zonas a "+z.getTipo());
    }
    public Zona subzona(int i){
        System.out.println("Este elemento no contiene subzonas");
    }
}
```

Bien, ya tenemos todos los tipos de zonas definidos, ahora toca crear un cliente para probar la funcionalidad:

```
public static void main(String[] args){
    // Creamos una lista de zonas
    ArrayList zonas = new ArrayList();
    Zona madrid = new Ciudad("Madrid");
    Zona vallecas = new Barrio("Vallecas");
    Zona chamberi = new Barrio("Chamberí");
    Zona hortaleza = new Barrio("Hortaleza");
    vallecas.agregarZona(new Calle("Alameda del Valle"));
    vallecas.agregarZona(new Travesia("Gavia"));
    chamberi.agregarZona(new Avenida("Reina Victoria"));
    chamberi.agregarZona(new Plaza("Colón"));
    hortaleza.agregarZona(new Travesia("Biosca"));
    hortaleza.agregarZona(new Avenida("América"));
    madrid.agregarZona(vallecas);
    madrid.agregarZona(chamberi);
    madrid.agregarZona(hortaleza);
    zonas.add(madrid);
    zonas.add(new Ciudad("Barcelona"));
    zonas.add(new Barrio("Georgetown"));
    zonas.add(new Calle("Aleatoria"));
    // Como veis, tenemos zonas de varios tipos y varios niveles
    // Generaremos todos los planos independientemente del nivel
    for(int i=0; i<zonas.size();i++){
        zonas.get(i).generarPlano();
    }
}
```

DECORATOR

Cuándo usar

- Para cambiar dinámicamente la funcionalidad de un objeto en tiempo de ejecución sin impactando la funcionalidad existente de los objetos.
- Para agregar funcionalidades que pueden retirarse más tarde.
- Para combinar múltiples funcionalidades donde no es práctico crear un subclase para cada combinación posible.

Intención

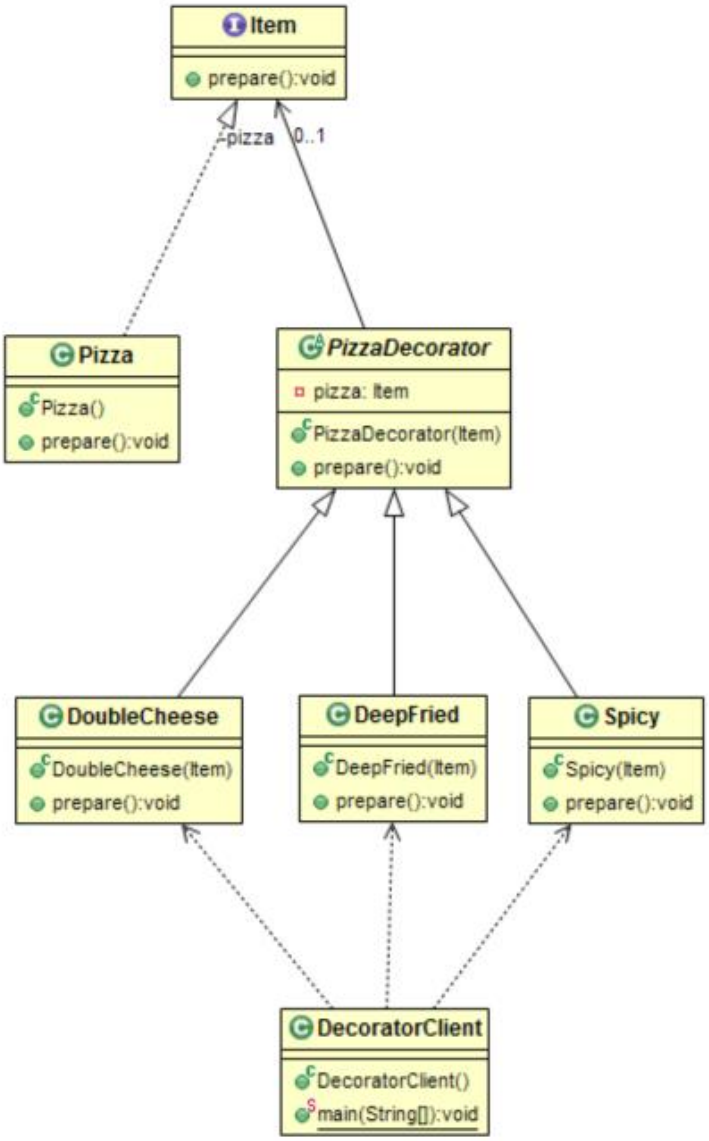
Asignar responsabilidades adicionales a un objeto dinámicamente Decoradores

Proporcionar una alternativa flexible a la subelasificación para ampliar la funcionalidad.

Problema

Suponga que necesita preparar una pizza que puede tener múltiples combinaciones de coberturas, tipo de cocción, etc. Aunque esto se puede lograr por herencia, no es práctico crear subclases para cada posible combinación. Alternativamente, puede usar Composición y agregar el requerido funcionalidades Dado que todas las implementaciones concretas se ajustan a lo mismo interfaz, podemos mezclar y combinar cualquier número de clases para crear un Variedad de combinaciones.

Estructura



Implementación

Comenzaremos definiendo nuestra interfaz Component que representará a los hoteles y a los futuros tipos de alojamiento que se ofertarán en el futuro, la llamaremos alquilable:

```
public interface Alquilable {

    public String getDescripcion();

    public String getTipo();

    public float obtenerPresupuesto();

}
```

A continuación definiremos una clase Hotel que sea Alquilable y que, evidentemente, representará los hoteles que ofertamos en nuestro buscador. Nuestro hotel tendrá un precio base de 100€ por noche:

```
public class Hotel implements Alquilable{
    private double coste_base = 100;
    private String tipo = "Hotel";
    private String descripcion;
    public Hotel(String descripcion){
        this.descripcion = descripcion;
    }
    public String getDescripcion(){
        return this.descripcion;
    }
    public String getTipo(){
        return this.tipo;
    }
    public float obtenerPresupuesto(){
        return this.coste_base;
    }
}
```

Ahora crearemos un Decorator que nos permitirá modificar el comportamiento de nuestro elemento Alquilable en tiempo de ejecución:

```
public abstract class AlquilableDecorator implements Alquilable{
    private Alquilable alquilable;
    public AlquilableDecorator(Alquilable alquilable){
        this.alquilable = alquilable;
    }
    public Alquilable getAlquilable(){
        return this.alquilable;
    }
```

```
public void setAlquilable(Alquilable alquilable){
        this.alquilable = alquilable;
    }
    public String getDescripcion(){
        return getAlquilable().getDescripcion();
    }
    public String obtenerTipo(){
        return getAlquilable().getTipo();
    }
    public float obtenerPresupuesto(){
        return getAlquilable().obtenerPresupuesto();
    }
}
```

Bien, ahora definiremos los complementos con los que extenderemos nuestra clase hotel los cuales podremos asignar en tiempo de ejecución (decoradores concretos):

```
public class PrimeraLineaDePlaya extends AlquilableDecorator{
    public PrimeraLineaDePlaya(Alquilable alquilable){
        super(alquilable);
    }
    public String getDescripcion(){
        return getAlquilable().getDescripcion().concat(" (vistas al mar)");
    }
    public float obtenerPresupuesto(){
        return getAlquilable().obtenerPresupuesto() + 100;
    }
}
public class PensionCompleta extends AlquilableDecorator{
    public PensionCompleta(Alquilable alquilable){
        super(alquilable);}
```

```
public String getDescripcion(){
        return getAlquilable().getDescripcion().concat(" (pension completa)");
    }
public float obtenerPresupuesto(){
return getAlquilable().obtenerPresupuesto() + 65;
}
}
public class DescuentoClienteVIP extends AlquilableDecorator{
    public DescuentoClienteVIP(Alquilable alquilable){
        super(alquilable);
    }
    public String getDescripcion(){
        return getAlquilable().getDescripcion().concat(" (descuento cliente VIP)");
    }
    public float obtenerPresupuesto(){
        return getAlquilable().obtenerPresupuesto() * 0.85;
    }
}
```

Y por último vamos a ver como utilizar todos nuestros ingredientes desde nuestro buscador, que será el cliente:

```
public static void main(String[] args){
    // Buscaremos un hotel en Torremolinos con pensión completa
    Alquilable hotel_torremolinos = new Hotel("Hotel en Torremolinos (Málaga)");
    hotel_torremolinos = new PensionCompleta(hotel_torremolinos);
    // Visualizamos el resultado
    System.out.println(hotel_torremolinos.getDescripcion());
    // Que mostrará: "Hotel en Torremolinos (Málaga) (pension completa)"
    // Obtenemos el presupuesto
    System.out.println(hotel_torremolinos.obtenerPresupuesto()+" €");
```

```
  // Que mostrará: "165 €"
        // Ahora buscaremos un hotel en Denia en primera linea de playa,
        // con pensión completa y le aplicaremos el descuento VIP
        Alquilable hotel_denia = new Hotel("Hotel en Denia (Alicante)");
        hotel_denia = new PrimeraLineaDePlaya(hotel_denia);
        hotel_denia = new PensionCompleta(hotel_denia);
        hotel_denia = new DescuentoClienteVIP(hotel_denia);
// Visualizamos el resultado
        System.out.println(hotel_torremolinos.getDescripcion());
        // Que mostrará: "Hotel en Denia (Alicante) (vistas al mar) (pension completa) (descuento cliente VIP)"
        // Obtenemos el presupuesto
        System.out.println(hotel_torremolinos.obtenerPresupuesto()+" €");
        // Que mostrará: "225.25 €"
}
```

EXTENSION OBJECT

Intención

Adjunte métodos adicionales a una clase. Mientras que Decorator requiere que la interfaz de la clase principal permanezca fija a medida que se aplican sucesivas "envolturas", los Objetos de extensión permiten que la interfaz de la clase crezca de forma incremental y dinámica.

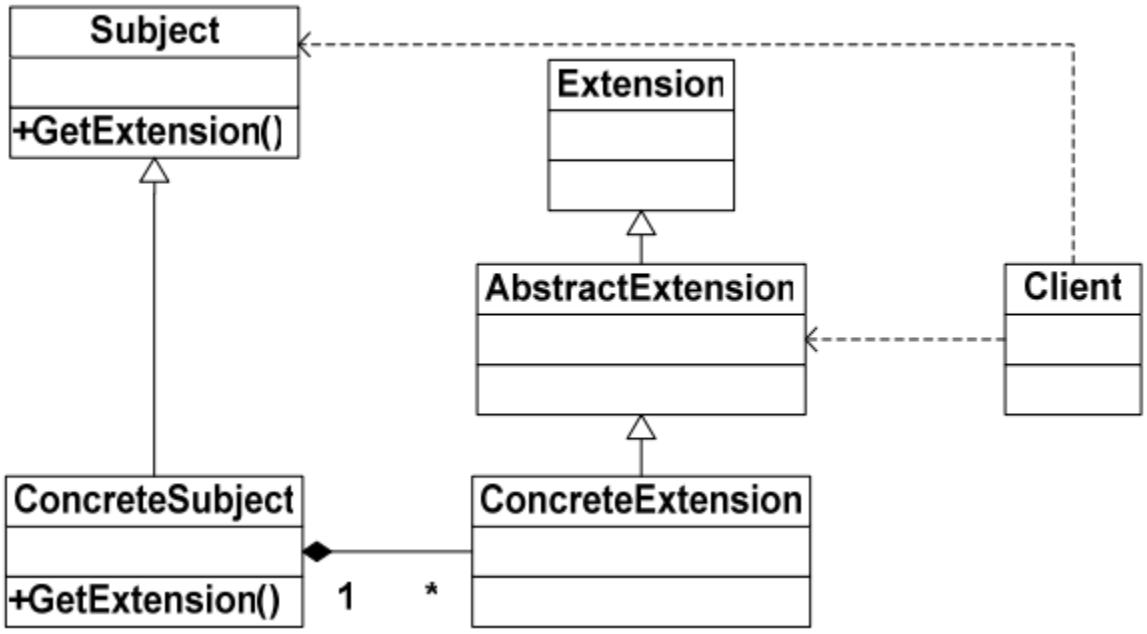
Implementación

Un puntero "hasa" de clase base de extensión abstracta a su propietario de clase base de sujeto abstracto. Las clases derivadas de Concrete Extension especifican el comportamiento (es decir, la interfaz) apropiado para sus responsabilidades y utilizan el miembro "propietario" para devolver la llamada al objeto Asunto correcto cuando se solicita su servicio.

Cada puntero concreto de la clase derivada de Sujeto "hasa" a una o más clases de Extensión que es significativo para él. También tiene un método "Extension * getExtension (char * type)" que acepta el identificador de la clase de extensión deseada y devuelve un puntero a esa clase si es compatible, o NULL si no lo es.

Finalmente, para acceder al objeto Extension de un objeto Subject, el cliente: llama al método getExtension con el "tipo" de la clase Extension deseada, realiza RTTI para probar el tipo del objeto devuelto, y si el resultado no es NULL, entonces el deseado Se llama al método en el objeto devuelto.

Modelo



A Subject class would be declared like this in C++:

```
class Subject {
public:
 //...
 virtual Extension* GetExtension(const char* name);
};
```

Subject::GetExtension is implemented as:

```
Extension* Subject::GetExtension(const char* name)
{
 return 0;
}
```

Here is a ConcreteSubject that provides a SpecificExtension:

```
class ConcreteSubject: public Subject {
```

```
public:
 //...
 virtual Extension* GetExtension(const char* name);
private:
 SpecificExtension* specificExtension;
};
```

The implementation of ConcreteSubject::GetExtension is defined like:

```
Extension* ConcreteSubject::GetExtension(const char* name)
{
 if (strcmp(name, "SpecificExtension" == 0) {
 if (specificExtension == 0)
 specificExtension = new SpecificExtension(this);
```



```
 return specificExtension;
 }
 return Subject::GetExtension(name);
}
```

Finally, to access an extension the client writes:

```
SpecificExtension* extension;
Subject* subject;
extension = dynamic_cast<SpecificExtension*>(
 subject->GetExtension("SpecificExtension")
);
if (extension) {
 // use the extension interface
}
```

FACADE

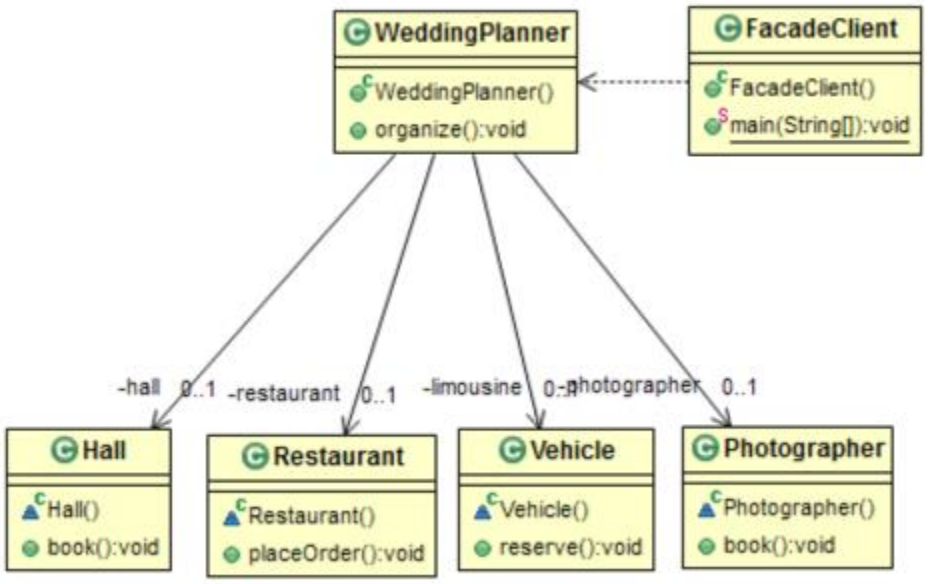
Cuándo usar

- Para proporcionar una interfaz simplificada a la funcionalidad general de un complejo subsistema.
- Promover la independencia y portabilidad del subsistema.

Intención

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

Estructura



Implementación

Supongamos que tenemos un conjunto de interfaces para un sistema que incluye muchos subsistemas La aplicación cliente puede usar estas interfaces para realizar. La operación requerida. Pero cuando la complejidad aumenta, el cliente la aplicación tendrá dificultades para administrarlo.

Al usar el patrón Fachada, podemos ocultar las complejidades del sistema y proporcionar una interfaz para el cliente mediante el cual el cliente puede acceder al sistema.

```
package com.genbetadev;

 public class Impresora {

                private String tipoDocumento;

                private String hoja;

                private boolean color;

                private String texto;

                public String getTipoDocumento() {

                                return tipoDocumento;

                 }

                 public void setTipoDocumento(String tipoDocumento) {

                                this.tipoDocumento = tipoDocumento;

                }

                public void setHoja(String hoja) {
                                this.hoja = hoja;
                }

                public String getHoja() {
```

```
                return hoja;

}

public void setColor(boolean color) {
                this.color = color;
}

public boolean getColor() {

                return color;

}

public void setTexto(String texto) {
                this.texto = texto;
}

public String getTexto() {

                return texto;

}


public void imprimir() {

                impresora.imprimirDocumento();
```

```
        }
}
```

Se trata de una clase sencilla que imprime documentos en uno u otro formato. El código de la clase cliente nos ayudará a entender mejor su funcionamiento.

```
package com.genbetadev;

public class PrincipalCliente {

        public static void main(String[] args) {

                Impresora i = new Impresora();

                i.setHoja("a4");

                i.setColor(true);

                i.setTipoDocumento("pdf");

                i.setTexto("texto 1");

                i.imprimirDocumento();

                Impresora i2 = new Impresora();

                i2.setHoja("a4");
```

```
                i2.setColor(true);

                i2.setTipoDocumento("pdf");

                i2.setTexto("texto 2");

                i2.imprimirDocumento();

                Impresora i3 = new Impresora();

                i3.setHoja("a3");

                i3.setColor(false);

                i3.setTipoDocumento("excel");

                i3.setTexto("texto 3");

                i3.imprimirDocumento();

        }

}
```

Como podemos ver la clase cliente se encarga de invocar a la impresora, y configurarla para después imprimir varios documentos .Ahora bien prácticamente todos los documentos que escribimos tienen la misma estructura (formato A4, Color , PDF). Estamos continuamente repitiendo código. Vamos a construir una nueva clase FachadaImpresoraNormal que simplifique la impresión de documentos que sean los más habituales.

```
package com.genbetadev;

public class FachadaImpresoraNormal
```

```
Impresora impresora;
                public FachadaImpresoraNormal(String texto) {
                                super();
                                impresora= new Impresora();
                                impresora.setColor(true);
                                impresora.setHoja("A4");
                                impresora.setTipoDocumento("PDF");
                                impresora.setTexto(texto);
                 }
                public void imprimir() {
                                impresora.imprimirDocumento();
                 }
}
```

De esta forma el cliente quedará mucho más sencillo :

```
 package com.genbetadev;
 public class PrincipalCliente2 {
                public static void main(String[] args) {
FachadaImpresoraNormal fachada1= new FachadaImpresoraNormal("texto1");
                                fachada1.imprimir();
                                FachadaImpresoraNormal fachada2= new
FachadaImpresoraNormal("texto2");
                                fachada2.imprimir();
                                Impresora i3 = new Impresora();
                                i3.setHoja("a4");
                                i3.setColor(true);
                                i3.setTipoDocumento("excel");
                                i3.setTexto("texto 3");i3.imprimirDocumento();

                } }
```

FLYWEIGHT

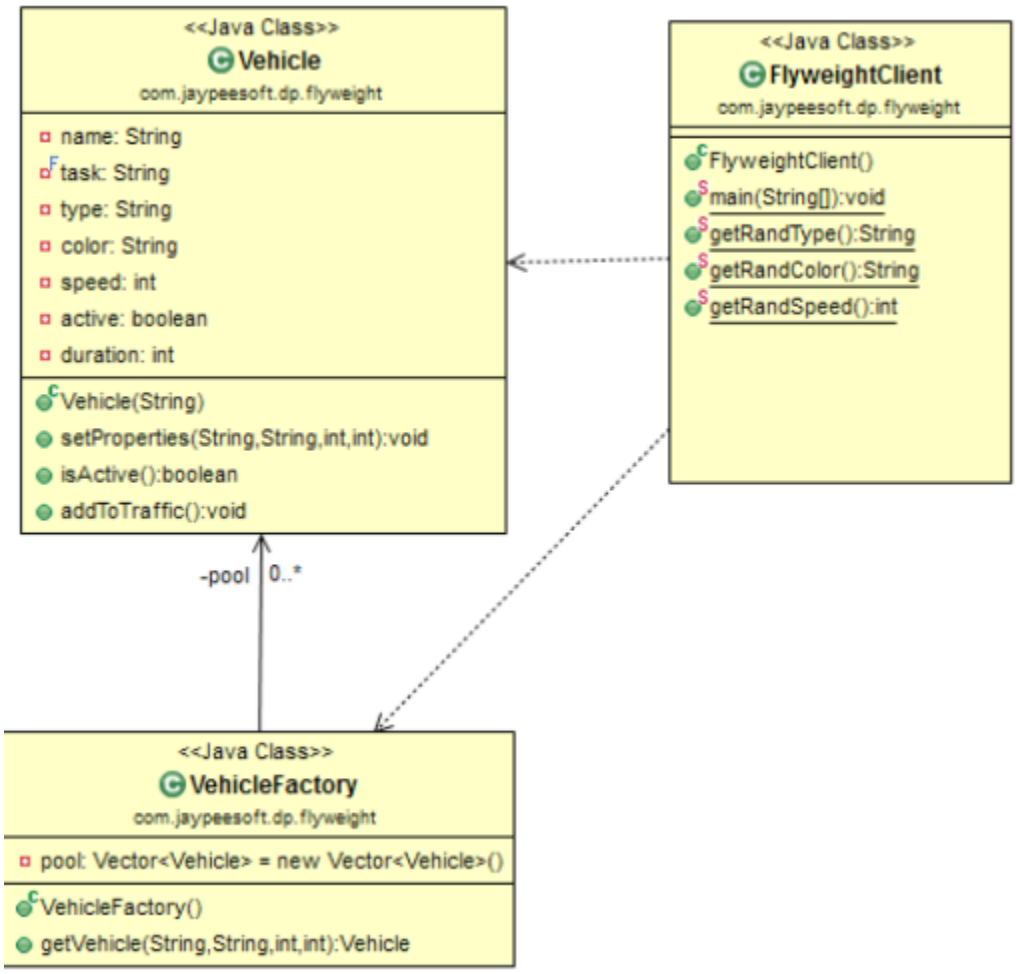
Cuándo usar

- Para mejorar el rendimiento cuando se necesita un gran número de objetos creado.
- Cuando la mayoría de los atributos del objeto pueden hacerse externos y compartidos.

Intención

Utilice el uso compartido para admitir grandes cantidades de objetos de grano fino de manera eficiente.

Estructura



Implementación

Vamos a aplicar lo visto en un ejemplo. Este patrón es muy utilizado en videojuegos, ya que nos permite que elementos con muchas características en común, como pueden ser los árboles del escenario, puedan ser reproducidos sin despilfarrar la memoria que gastarían haciendo una instancia por cada uno.

Para desarrollar el escenario anterior vamos a definir en primer lugar la interfaz Flyweight. Tendremos determinados tipos de árboles que se dibujarán centenas de veces a lo largo de nuestros escenarios, por lo que nos interesa la característica extrínseca tipo.

```
public interface IArbolLigero
{
    public String getTipo();
    public void dibujar( long x, long y, long z );
}
```

Ahora crearemos el ConcreteFlyweight, que en este caso representará al arbol concreto (definiendo las características intrínsecas del mismo):

```
public class Arbol implements IArbolLigero
{
    private String tipo;
    public Arbol( String tipo )
    {
        this.tipo = tipo;
    }
    @Override
    public String getTipo()
    {
        return this.tipo;
    }
    @Override
    public void dibujar( long x, long y, long z )
    {
```

```
System.out.println("Árbol [" + this.getTipo() + "] dibujado en las coordenadas ("+x+", "+y+", "+z+")" );
    }
}
```

El patrón Flyweight combina un Singleton con una Factoría para lograr proporcionar un punto de acceso único a la clase desde el cual se puedan reutilizar los elementos no redundantes. Por tanto, nuestra FlyweightFactory quedará definida como:

```
public class FabricaDeArboles
{
    private Map arboles;
    public FabricaDeArboles()
    {
        this.arboles = new HashMap();
    }
public IArbolLigero getArbol( String tipo )
    {
        // Si no tenemos ningún árbol del tipo solicitado, lo creamos
        if(!arboles.containsKey(tipo))
        {
            arboles.put(tipo, new Arbol(tipo));
        }
        // Devolvemos el árbol del tipo correspondiente
        return arboles.get(tipo);
    }
}
```

Una vez definida la fábrica de árboles ya tenemos todos los ingredientes. Definiremos ahora nuestro cliente como si fuera el motor de un videojuego que tiene que llenar un paisaje de pinos, abetos y sauces:

```
public static void main(String args)
{
// Obtenemos el número de árboles a dibujar por parámetro
```

```
int num_arboles = Integer.PatronesInt(args[0]);
// Definimos los tipos de árbol
String[] tipos = {"pino","abeto","sauce"};
// Creamos la fábrica de Árboles
FabricaDeArboles f = new FabricaDeArboles();
// Crearemos tantos árboles como se indiquen por parámetro
// El tipo de árbol será seleccionado aleatoriamente
Random r = new Random(tipos.length),
p = new Random();
for(int i=0;i<num_arboles;i++)
    {
        f.getArbol(tipos[r.nextInt()])
         .dibujar(
              p.nextLong(),
              p.nextLong(),
              p.nextLong()
        );
    }
}
```

FRONT CONTROLLER

¿Qué es?

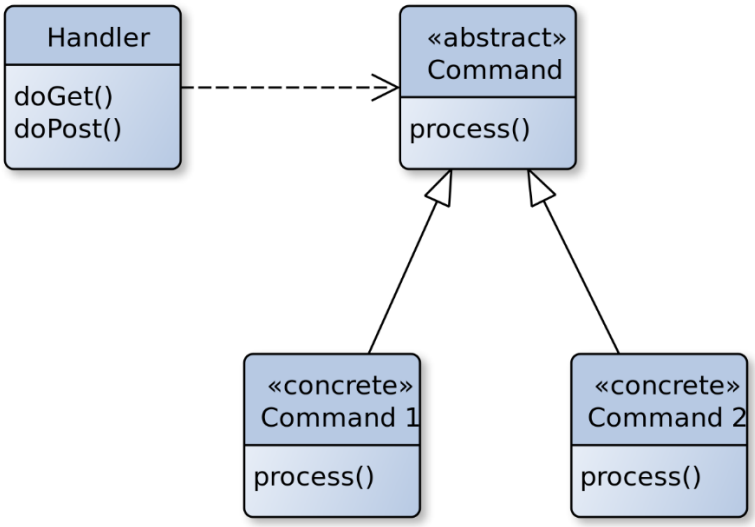
El patrón de diseño del software del controlador frontal aparece en varios catálogos de patrones y está relacionado con el diseño de aplicaciones web. Es "un controlador que maneja todas las solicitudes de un sitio web", que es una estructura útil para que los desarrolladores de aplicaciones web logren la flexibilidad y la reutilización sin redundancia de código.

Instrucción

Una típica estructura de controlador frontal.

Los controladores frontales a menudo se usan en aplicaciones web para implementar flujos de trabajo. Si bien no es estrictamente necesario, es mucho más fácil controlar la navegación a través de un conjunto de páginas relacionadas (por ejemplo, varias páginas utilizadas en una compra en línea) desde un controlador frontal que hacer que las páginas individuales sean responsables de la navegación.

Estructura



Demo implementation in Java

Here is part of a demo code to implement front controller.

```
private void doProcess(HttpServletRequest request,
                       HttpServletResponse response)
   throws IOException, ServletException {
   try {
      getRequestProcessor().processRequest(request);
      getScreenFlowManager().forwardToNextScreen(request, response);
   } catch (Throwable ex) {
      String className = ex.getClass().getName();
      nextScreen = getScreenFlowManager().getExceptionScreen(ex);
      // Put the exception in the request
      request.setAttribute("javax.servlet.jsp.jspException", ex);
      if (nextScreen == null) {
         // Send to general error screen
         ex.printStackTrace();
         throw new ServletException("MainServlet: unknown exception: " +
            className);  }  }
```

PROXY

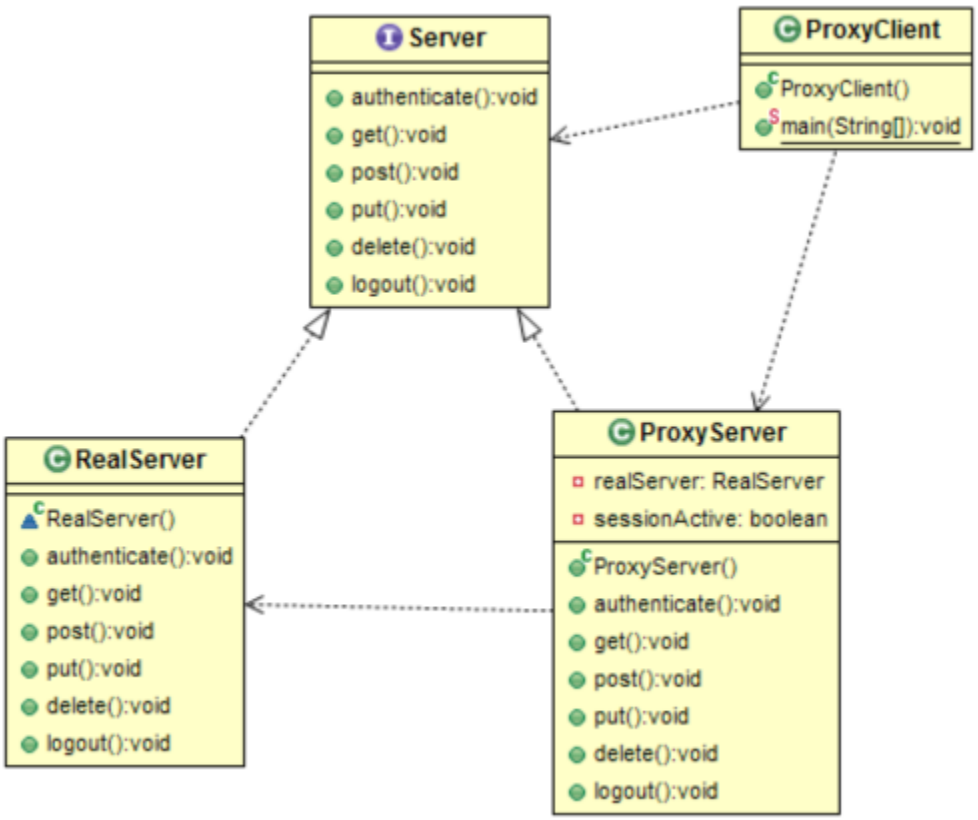
Cuándo usar

- Para proporcionar acceso controlado a un objeto maestro sensible Para proporcionar una referencia local a un objeto remoto
- Para mejorar el rendimiento cuando se necesita acceder a un objeto frecuentemente

Intención

Proporcionar un sustituto o marcador de posición para otro objeto para controlar el acceso a eso.

Estructura



Ejemplo:

Toca clarificar los conceptos mediante un ejemplo. En este caso vamos a hacer un ejemplo clásico de control de acceso mediante un proxy de protección. Para ello vamos a imaginar un escenario en el que se desea realizar la descarga de cierto material que, pese a estar disponible públicamente, no está permitida su posesión en el país del cliente.

En primer lugar definimos la interfaz del servidor de descargas:

```
public interface Server{
    public void download(String url);
}
```

A continuación vamos a implementar nuestro servidor real:

```
public class RealServer implements Server{
    private int port;
    private String host;
    public RealServer(int port, String host){
        this.port=port;
        this.host=host;
        System.out.println("Servidor iniciado...");
    }
    public void download(String url){
        System.out.println("Descargando "+host+":"+port+"/"+url);
    }
}
```

Ahora vamos a implementar el proxy que incluye el control de acceso restringido:

```
public class ProxyServer implements Server{
    RealServer srv;
    private int port;
    private String host;
    public ProxyServer(int port, String host){
        this.port=port;
        this.host=host;
        srv=null;
        System.out.println("Proxy iniciado...");
    }
```

```
public void download(String url){
if(!isRestricted(url))
{
if(srv = null)
{
srv = new RealServer(port,host);
  }
srv.download(url);
}else{
System.out.println("Actualmente se encuentra en un área que no permite la descarga del fichero.");
} }
public boolean isRestricted(String fichero) {
return !fichero.equals("/descarga/prohibida.avi");
} }
```

Puesto que no es relevante para el ejemplo, la implementación del método isRestrictedArea será dummy. Ya tenemos todos los ingredientes, así que vamos a probarlo todo mediante el método main de nuestro cliente:

```
public static void main(String [] args){
    // Creamos el proxy a la página de descargas
    Server srv = new ProxyServer(20,"http://paginadedescarg.as");
    // Descargamos un archivo permitido
    srv.download("/descarga/permitida.avi");
    // En este punto será donde se cree el objeto RealServer
    // Vamos a probar ahora con una descarga restringida
    srv.download("/descarga/prohibida.avi");
}
```

DISEÑO DE SOFTWARE – COMPORTAMIENTO

CHAIN OF RESPONSABILITY

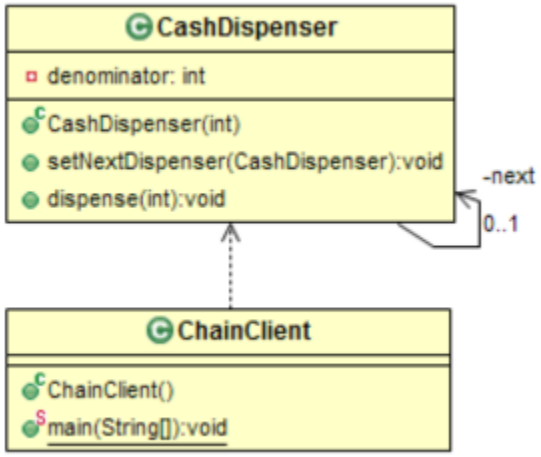
Cuándo usar

- Cuando una solicitud necesita ser procesada por múltiples procesadores
- Para lograr un acoplamiento flojo entre el emisor y los receptores

Intención

Evite acoplar el remitente de una solicitud a su receptor dando más de un objeto la oportunidad de manejar la solicitud. Encadene los objetos receptores y pase la solicitud a lo largo de la cadena hasta que un objeto la maneje.

Estructura




```
public class Cliente {

   public static void main(String argv[]) {

     Unidad smith  = new Coronel("Smith", null);

     Unidad truman = new Coronel("Truman", "Tomar posición enemiga");

     Unidad ryan   = new Soldado("Ryan");

     Unidad rambo  = new Soldado("Rambo");

     System.out.println(rambo.orden());    // rambo ->

     rambo.establecerMando(truman);

     System.out.println(rambo.orden());    // rambo -> truman
```

```
    ryan.establecerMando(rambo);
     System.out.println(ryan.orden());     // ryan -> rambo -> truman
    }
}
/**
 * La clase Unidad representa la clase abstracta manejadora de la cadena de responsabilidad.
 * El servicio delegado en la cadena es la solicitud de una orden al mando directo
 */
public abstract class Unidad {
    /* en el constructor, además de un nombre para la unidad, se inicializa la referencia
       que implementa la cadena de responsabilidad (_mando): en principio no hay sucesor */
    public Unidad(String nombre) {
        _mando = null;
        _nombre = nombre;
    }
    public String toString() { return _nombre; }
    // cambia el mando de una unidad (modifica cadena de responsabilidad)
    public void establecerMando(Unidad mando) { _mando = mando; }
    /* comportamiento por defecto de la cadena: delegar en el mando directo o, si se
       alcanza el final de la cadena, utilizar una resolución por defecto (sin orden) */
    public String orden() {
        return (_mando != null ? _mando.orden() : "(sin orden)");
    }
    private Unidad _mando;
    private String _nombre;
}
/**
 * La clase Coronel modifica ligeramente el comportamiento por defecto de la cadena de
 * responsabilidad: si el coronel tiene una orden específica, utiliza ésta para resolver
```

```
 * el servicio. Si no tiene una orden específica (_orden==null), emplea el comportamiento
 * convencional de las unidades
 */
public class Coronel extends Unidad {
   // inicializa la parte de unidad e inicializa el estado propio del Coronel (_orden)
   public Coronel(String nombre, String orden) {
     super(nombre);
     _orden = orden;
   }
   /* refinamiento del servicio que utiliza la cadena de responsabilidad, resolviendo
      localmente si tiene órdenes específicas o comportándose convencionalmente en
      caso contrario */
   public String orden()    { return (_orden != null ? _orden : super.orden()); }
   public String toString() { return ("Coronel " + super.toString());
   private String _orden;
}
/**
 * Esta clase es una extensión instanciable de la superclase Unidad que respeta el
 * comportamiento por defecto de la cadena de responsabilidad
 */
public class Soldado extends Unidad {
   // el constructor sólo tiene que inicializar la parte correspondiente a la superclase
   public Soldado(String nombre) {
      super(nombre);
   }
   public String toString() { return ("Soldado " + super.toString()); }
}
```

COMMANDO

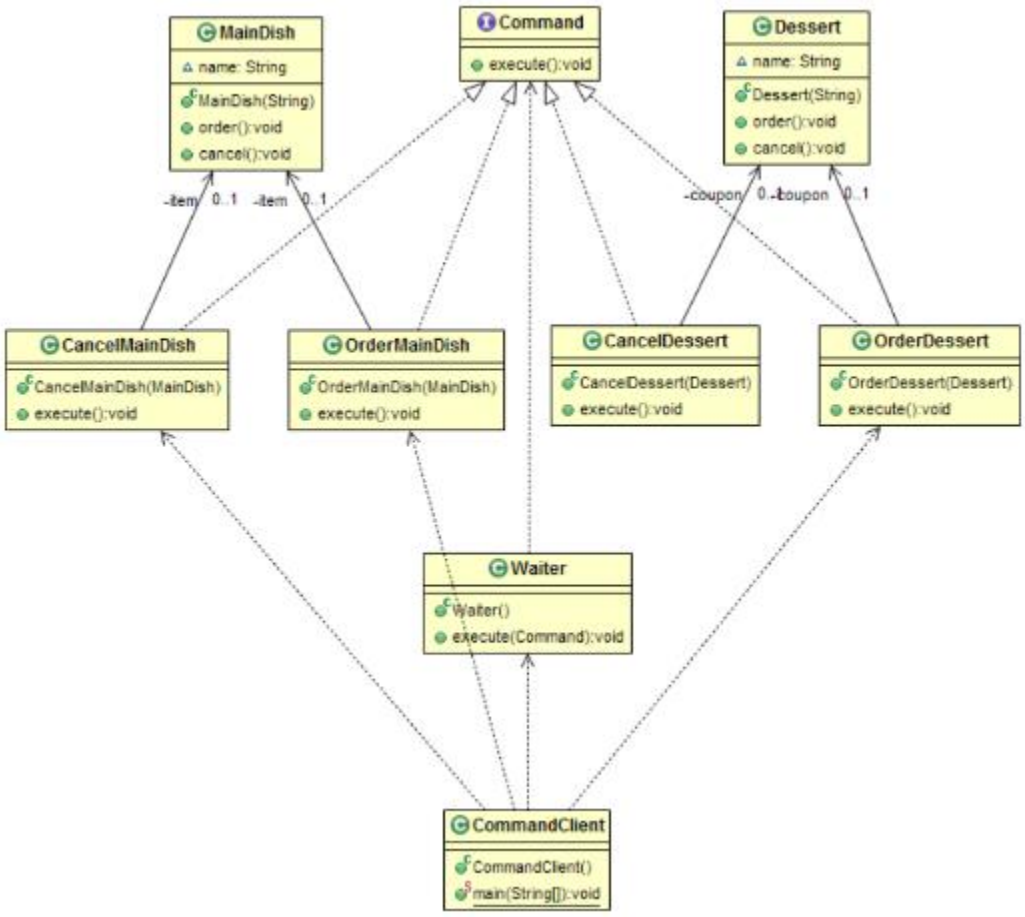
Cuándo usar

- Cuando el ejecutor del comando no necesita saber nada en todo sobre qué es el comando, qué información de contexto necesita o que hace.
- Para registrar una devolución de llamada cuando se activa algún evento.

Intención

Encapsula una solicitud como un objeto, lo que te permite parametrizar clientes con diferentes solicitudes, solicitudes de cola o registro, y soporte para deshacer operaciones

Estructura



Ejemplo:

Veamos un simple ejemplo de uso del patrón Command. Para ello vamos a plantear el escenario donde estamos implementando el control de nuestra vivienda inteligente mediante una aplicación para el móvil:

```
// En primer lugar definiremos la interfaz Command

public interface Command{

    public void execute();

}

// Ahora definiremos una serie de clases que implementarán la funcionalidad

// correspondiente al elemento de nuestra casa inteligente que queremos

// utilizar (Receivers).

// Elemento que permite apagar y encender las luces

public class Luces{

    public boolean conectar(){

        System.out.println("Conectando al sistema de iluminación...");

        try{

            System.out.println("Conexión al sistema de iluminación establecida.");

            return true;

        }catch(Exception e){

            System.out.println("No se ha podido establecer la conexión al sistema de iluminación.
ERROR:n"+e.getMessage());

            return false;

        }

    }

    public boolean desconectar(){

        System.out.println("Desconectando del sistema de iluminación...");

        try{

            System.out.println("Se ha desconectado del sistema de iluminación.");

            return true;

        }catch(Exception e){
```

```
System.out.println("No se ha podido desconectar del sistema de iluminación.
ERROR:n"+e.getMessage());
                return false;
            }
        }
        public boolean encender(){
            System.out.println("Encendiendo el sistema de iluminación...");
            try{
                System.out.println("Sistema de iluminación encendido.");
                return true;
            }catch(Exception e){
                System.out.println("No se ha podido encender el sistema de iluminación.
ERROR:n"+e.getMessage());
                return false;
            }
        }
        public boolean apagar(){
            System.out.println("Apagando el sistema de iluminación...");
            try{
                System.out.println("Sistema de iluminación apagado.");
                return true;
            }catch(Exception e){
                System.out.println("No se ha podido apagar el sistema de iluminación.
ERROR:n"+e.getMessage());
                return false;
            }
        }
}
// Elemento que permite abrir y cerrar la portada
public class Portada{
```

```
public boolean conectar(){
System.out.println("Conectando al sistema de la portada...");
try{
System.out.println("Conexión al sistema de la portada establecida.");
return true;
}catch(Exception e){
System.out.println("No se ha podido establecer la conexión al sistema de la portada.
ERROR:n"+e.getMessage());
return false;
  }
}
public boolean desconectar(){
System.out.println("Desconectando del sistema de la portada...");
try{
System.out.println("Se ha desconectado del sistema de la portada.");
return true;
}catch(Exception e){
System.out.println("No se ha podido desconectar del sistema de la portada. ERROR:n"+e.getMessage());
return false;
} }
public boolean abrir(){
System.out.println("Abriendo la portada...");
try{
System.out.println("Portada abierta.");
return true;
}catch(Exception e){
System.out.println("No se ha podido abrir la portada. ERROR:n"+e.getMessage());
return false;
            }  }
```

```
public boolean cerrar(){
        System.out.println("Cerrando la portada...");
        try{
            System.out.println("Portada cerrada.");
            return true;
        }catch(Exception e){
            System.out.println("No se ha podido cerrar la portada. ERROR:n"+e.getMessage());
            return false;
        }
    }
}
// Ahora definiremos los comandos concretos para cada acción
// (Concrete Commands)
// Comando para encender las luces
public class EncenderLuces implements Command{
    private Luces luces.
public EncenderLuces(){
this.luces = new Luces(); }
    public void execute(){
        luces.conectar();
        luces.encender();
        luces.desconectar();
    }
}
// Comando para apagar las luces
public class ApagarLuces implements Command{
    private Luces luces.
  public ApagarLuces(){
this.luces = new Luces(); }
```

```
public void execute(){
 luces.conectar();
 luces.apagar();
luces.desconectar();
}
}
// Comando para abrir la portada
public class AbrirPortada implements Command{
	private Portada portada.
	public AbrirPortada(){
		this.portada = new Portada();
	}
	public void execute(){
		luces.conectar();
		luces.abrir();
		luces.desconectar();
	}
}
// Comando para cerrar la portada
public class CerrarPortada implements Command{
	private Portada portada.
	public CerrarPortada(){
		this.portada = new Portada();
	}
	public void execute(){
		luces.conectar();
		luces.cerrar();
		luces.desconectar();
	} }
```

```
// Ahora vamos a definir el Invoker, que simplemente será
// el encargado de llamar a una orden
public class Invoker{
	private Command orden;
	public Invoker(Command orden){
		this.orden = orden;
	}
	public void run(){
		orden.execute();
	}
}
// Vamos a ver el funcionamiento
public static void main(String [] args){
	Command command;
	if(args[0].equals("encender") && args[1].equals("luces")){
		command = new EncenderLuces();
	} else if(args[0].equals("apagar") && args[1].equals("luces")){
		command = new ApagarLuces();
	} else if(args[0].equals("abrir") && args[1].equals("portada")){
		command = new AbrirPortada();
	} else if(args[0].equals("cerrar") && args[1].equals("portada")){
		command = new CerrarPortada();
	}
	Invoker invoker = new Invoker(command);
	invoker.run();
}
```

INTERPRETER

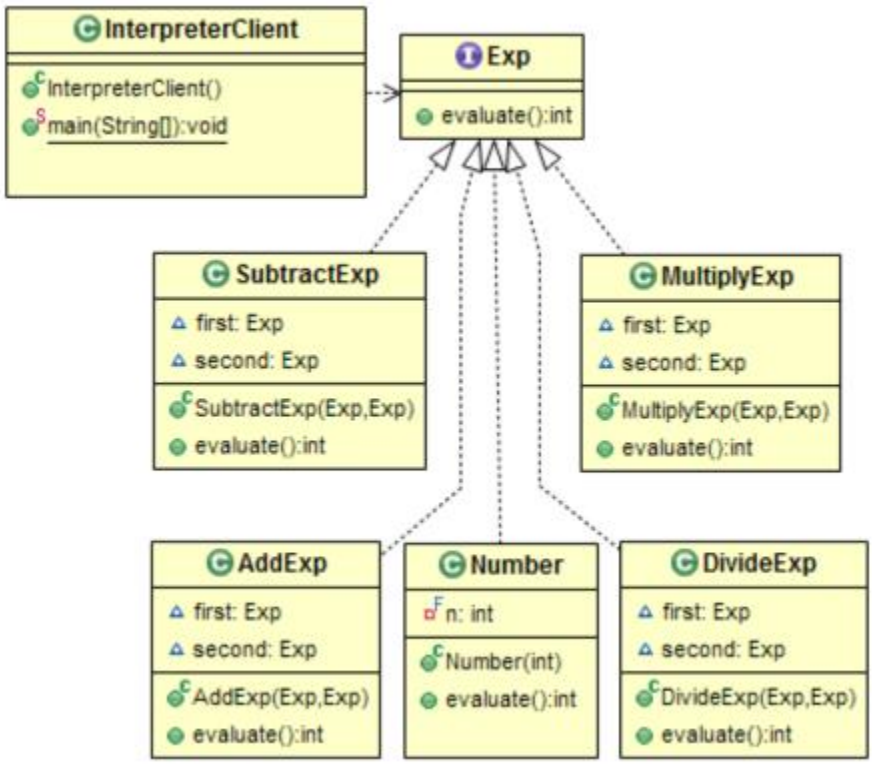
Cuándo usar

- Cuando el ejecutor del comando no necesita saber nada en todo sobre qué es el comando, qué información de contexto necesita o que hace.
- Para registrar una devolución de llamada cuando se activa algún evento.

Intención

Encapsula una solicitud como un objeto, lo que te permite parametrizar clientes con diferentes solicitudes, solicitudes de cola o registro, y soporte para deshacer operaciones

Estructura



Java

El siguiente ejemplo en Java muestra como un lenguaje de propósito general podría interpretar un lenguaje más especializado, aquí la Notación polaca inversa.

La salida es:

'42 2 1 - +' equals 43

```
import java.util.*;

interface Expression {
```

```
public void interpret(Stack<Integer> s);
}
class TerminalExpression_Number implements Expression {
  private int number;
  public TerminalExpression_Number(int number)      { this.number = number; }
  public void interpret(Stack<Integer> s)  { s.push(number); }
}
class TerminalExpression_Plus implements Expression {
  public void interpret(Stack<Integer> s)  { s.push( s.pop() + s.pop() ); }
}
class TerminalExpression_Minus implements Expression {
  public void interpret(Stack<Integer> s)  { int tmp = s.pop(); s.push( s.pop() - tmp ); }
}
class Parser {
  private ArrayList<Expression> parseTree = new ArrayList<Expression>(); // only one NonTerminal
Expression here
  public Parser(String s) {
    for (String token : s.split(" ")) {
      if     (token.equals("+")) parseTree.add( new TerminalExpression_Plus() );
      else if (token.equals("-")) parseTree.add( new TerminalExpression_Minus() );
      // ...
      else                    parseTree.add( new TerminalExpression_Number(Integer.valueOf(token)) );
    }
  }
  public int evaluate() {
    Stack<Integer> context = new Stack<Integer>();
    for (Expression e : parseTree) e.interpret(context);
    return context.pop();
  }
```

```
}
class InterpreterExample {
  public static void main(String[] args) {
    System.out.println("'42 2 1 - +' equals " + new Parser("42 2 1 - +").evaluate());
  }
}
```

ITERATOR

Cuándo usar

- Para proporcionar una forma estándar de atravesar colecciones de similares objetos.
- Exponer una interfaz simple al cliente ocultando las complejidades de el recorrido.
- Para proporcionar una interfaz uniforme para atravesar diferentes agregados estructuras

Intención

Proporcionar una forma de acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación subyacente.

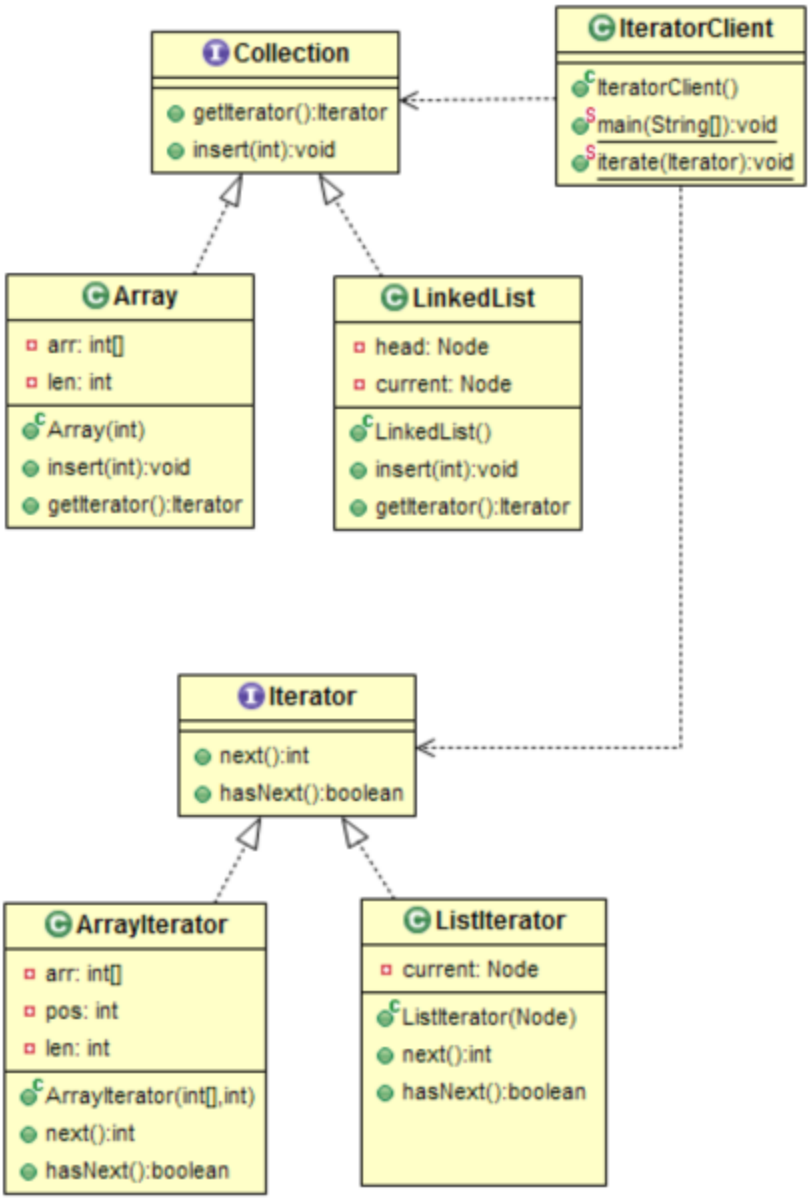
Componentes

1. Una interfaz para la colección.

2. Implementaciones concretas de la interfaz de colección.

3. Una clase "iterador" que puede encapsular el recorrido de la clase "colección".

Estructura



Para la creación del patrón iterador debe implementarse el control de la iteración (pudiendo ser un iterador externo que ofrece los métodos para que el cliente recorra la estructura paso a paso, o un iterador interno que ofrece un método de actuación sobre la estructura que, de manera transparente al cliente, la recorre aplicándose a todos sus elementos) y definirse el recorrido. A mayores se podrían implementar operaciones adicionales en el iterador o definir la estructura de éste de una manera más robusta ante cambios en la estructura. Hay que tener especial cuidado en la implementación de iteradores con accesos privilegiados, iteradores para estructuras compuestas o iteradores nulos.

```
public class Vector2 {
    public int[] _datos;
    public Vector2(int valores){
        _datos = new int[valores];
        for (int i = 0; i < _datos.length; i++){
            _datos[i] = 0;
        }
    }
    public int getValor(int pos){
        return _datos[pos];
    }
    public void setValor(int pos, int valor){
        _datos[pos] = valor;
    }
    public int dimension(){
        return _datos.length;
    }
    public IteradorVector iterador(){
        return new IteradorVector(this);
    }
}
```

Definición del iterador concreto.

```
public class IteradorVector{
    private int[] _vector;
    private int _posicion;
    public IteradorVector(Vector2 vector) {
        _vector = vector._datos;
        _posicion = 0;
    }
```

```
public boolean hasNext(){
      if (_posicion < _vector.length)
                  return true;
            else
                  return false;
   }
   public Object next(){
            int valor = _vector[_posicion];
            _posicion++;
            return valor;
   }
}
```

Ejemplo de uso

```
   public static void main(String argv[]) {
            Vector2 vector = new Vector2(5);
            //Creación del iterador
            IteradorVector iterador = vector.iterador();
            //Recorrido con el iterador
            while (iterador.hasNext())
           System.out.println(iterador.next());
   }
```

En java ya existe una interfaz iterator que hace a las veces de abstracción. Todos los iteradores utilizados en sus librerías cumplen esta interfaz, lo que permite tratarlos a todos ellos de manera uniforme. Hacer que nuestro IteradorVector implementase Iterator permitiría que fuese utilizado por otras librerías y programas de manera transparente.

MEDIATOR

Cuándo usar

- Para facilitar las interacciones entre un conjunto de objetos donde el las comunicaciones son complejas y difíciles de mantener.
- Tener un control centralizado para las interacciones de objetos.
-

Intención

Defina un objeto que encapsule cómo interactúa un conjunto de objetos. Mediador promueve el acoplamiento flojo al evitar que los objetos se refieran entre sí explícitamente, y le permite variar su interacción de forma independiente.

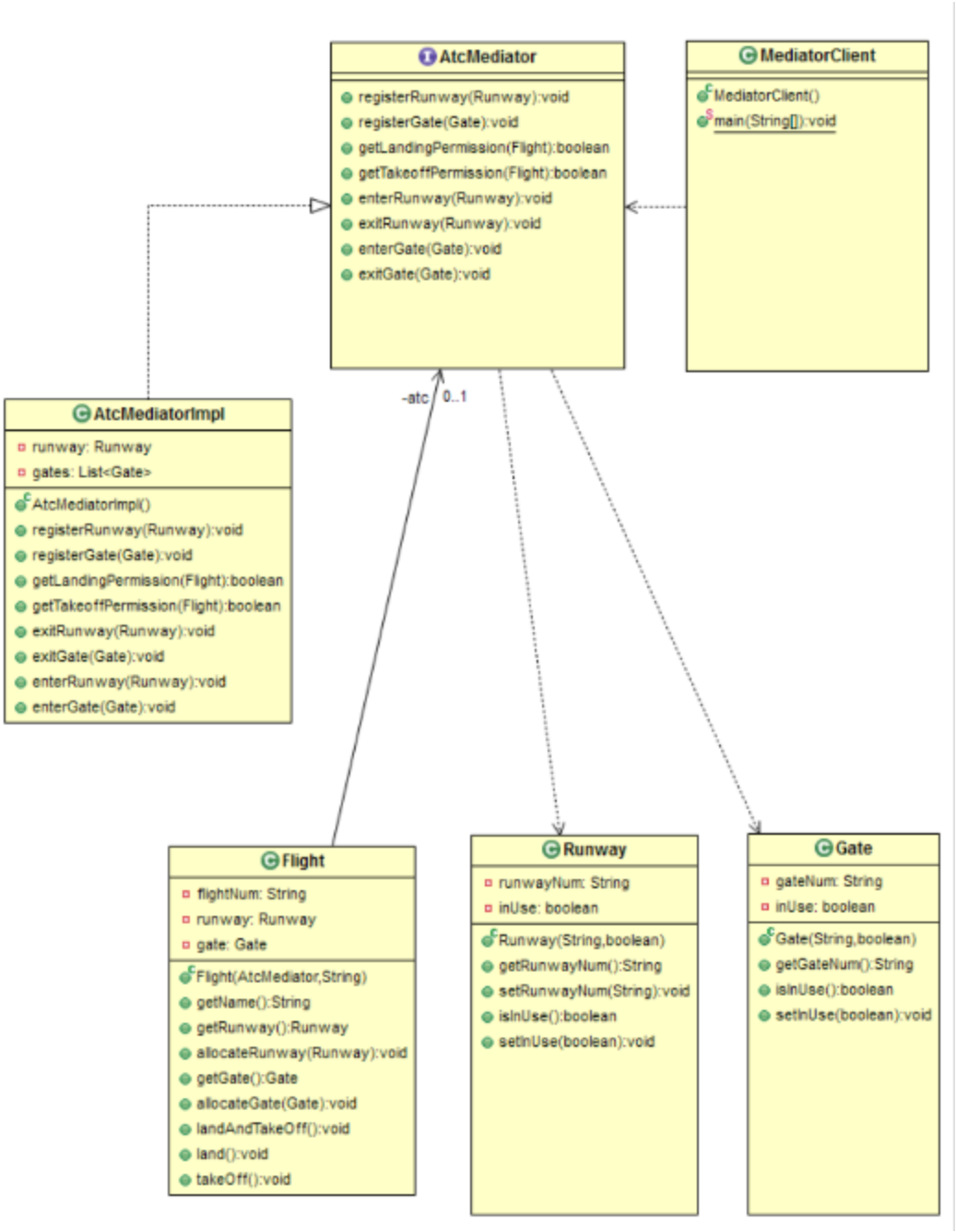
Componentes

1. Interfaz de mediador: una interfaz que define las reglas de comunicación. entre objetos

2. Mediador concreto: un objeto mediador que permitirá comunicación entre objetos participantes

3. Colegas: objetos que se comunican entre sí a través del mediador. Objeto

Estructura



Implementación

```
import java.awt.Font;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;
```

```
import javax.swing.JLabel;
import javax.swing.JPanel;
//Interfaz Amigo
interface Command {
   void execute();
}
//Mediador Abstracto
interface IMediator {
   void book();
   void view();
   void search();
   void registerView(BtnView v);
   void registerSearch(BtnSearch s);
   void registerBook(BtnBook b);
   void registerDisplay(LblDisplay d);
}
//Mediador Concreto
class Mediator implements IMediator {
   BtnView btnView;
   BtnSearch btnSearch;
   BtnBook btnBook;
   LblDisplay show;
   //....
   void registerView(BtnView v) {
      btnView = v;
   }
   void registerSearch(BtnSearch s) {
      btnSearch = s;
   }
```

```
void registerBook(BtnBook b) {
        btnBook = b;
    }
    void registerDisplay(LblDisplay d) {
        show = d;
    }
    void book() {
        btnBook.setEnabled(false);
        btnView.setEnabled(true);
        btnSearch.setEnabled(true);
        show.setText("booking...");
    }
    void view() {
        btnView.setEnabled(false);
        btnSearch.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("viewing...");
    }
    void search() {
        btnSearch.setEnabled(false);
        btnView.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("searching...");
    }
}
//Un amigo concreto
class BtnView extends JButton implements Command {
    IMediator med;
```

```
  BtnView(ActionListener al, IMediator m) {
      super("View");
      addActionListener(al);
      med = m;
      med.registerView(this);
   }
   public void execute() {
      med.view();
}
//Un amigo concreto
class BtnSearch extends JButton implements Command {
   IMediator med;
   BtnSearch(ActionListener al, IMediator m) {
      super("Search");
      addActionListener(al);
      med = m;
      med.registerSearch(this);
   }
   public void execute() {
      med.search();
   }
}
//Un amigo concreto
class BtnBook extends JButton implements Command {
   IMediator med;
   BtnBook(ActionListener al, IMediator m) {
      super("Book");
      addActionListener(al);
      med = m;
```

```
  med.registerBook(this);
    }
    public void execute() {
        med.book();
    }
}
class LblDisplay extends JLabel {
    IMediator med;
    LblDisplay(IMediator m) {
        super("Just start...");
        med = m;
        med.registerDisplay(this);
        setFont(new Font("Arial", Font.BOLD, 24));
    }
}
class MediatorDemo extends JFrame implements ActionListener {
    IMediator med = new Mediator();
    MediatorDemo() {
        JPanel p = new JPanel();
        p.add(new BtnView(this, med));
        p.add(new BtnBook(this, med));
        p.add(new BtnSearch(this, med));
        getContentPane().add(new LblDisplay(med), "North");
        getContentPane().add(p, "South");
        setSize(400, 200);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent ae) {
        Command comd = (Command) ae.getSource();
```

```
    comd.execute();
  }
  public static void main(String[] args) {
    new MediatorDemo();  } }
```

MEMENTO

Cuándo usar

- Para tomar instantáneas y restaurar un objeto a su estado anterior (p. Ej. Operaciones de "deshacer" o "deshacer").

Intención

Sin violar la encapsulación, capture y externalice el objeto interno de un objeto. estado para que el objeto pueda restaurarse a este estado más adelante.

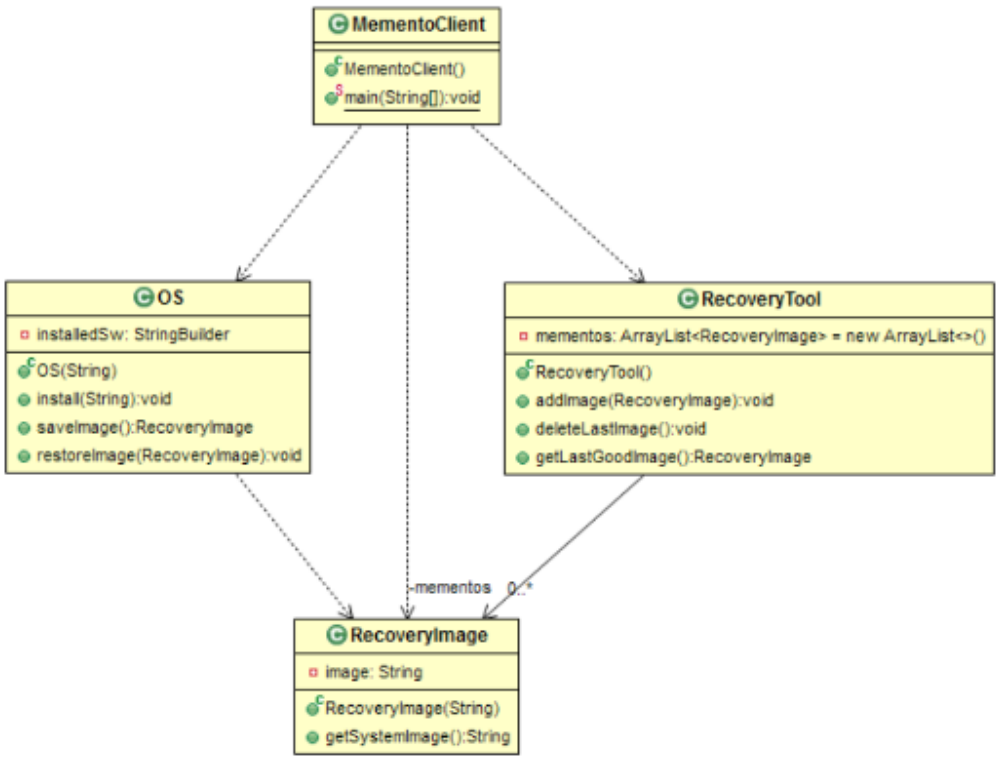
Componentes

1. Originador: el objeto que sabe cómo salvarse.

2. Cuidador: el objeto que sabe por qué y cuándo el Originador necesita guardar y restaurar a sí mismo.

3. Memento: la caja de bloqueo que escribe y lee el Originador, y pastoreado por el cuidador.

Estructura



Java

El siguiente programa Java ilustra el uso de "undo" con el patrón Memento. Intervienen tres clases: la clase Originator (Originador) es una clase que cambia de estado; la clase Caretaker (Portero) se encarga de registrar los cambios del Originador; Memento (Recuerdo) guarda el objeto a salvaguardar.

```
import java.util.*;

class Memento
{
   private String state;
   public Memento(String stateToSave)
   {
      state = stateToSave;
   }
   public String getSavedState()
   {
      return state;
   }
}
class Originator
{
   private String state;
   /* lots of memory consumptive private data that is not necessary to define the
    * state and should thus not be saved. Hence the small memento object. */
   public void set(String state)
   {
     System.out.println("Originator: Setting state to "+state);
     this.state = state;
   }
   public Memento saveToMemento()
   {
```

```
System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }
    public void restoreFromMemento(Memento m)
    {
        state = m.getSavedState();
        System.out.println("Originator: State after restoring from Memento: "+state);
    }
}
class Caretaker {
  private ArrayList<Memento> savedStates = new ArrayList<Memento>();
  public void addMemento(Memento m) { savedStates.add(m); }
  public Memento getMemento(int index) { return savedStates.get(index); }
}
class MementoExample {
  public static void main(String[] args) {
      Caretaker caretaker = new Caretaker();
      Originator originator = new Originator();
      originator.set("State1");
      originator.set("State2");
      caretaker.addMemento( originator.saveToMemento() );
      originator.set("State3");
      caretaker.addMemento( originator.saveToMemento() );
      originator.set("State4");
      originator.restoreFromMemento( caretaker.getMemento(1) );
  }
}
```

La salida en pantalla es:

Originator: Setting state to State1

Originator: Setting state to State2

Originator: Saving to Memento.

Originator: Setting state to State3

Originator: Saving to Memento.

Originator: Setting state to State4

Originator: State after restoring from Memento: State3

```
/**
 * Memento pattern: Copy the information into a another class for recovery in the future if necessary
 * @author Asenior
 *
 */
public class MementoPattern {
	public void main(String args[]){
		RegularClass regularClass = new RegularClass();
		regularClass.setData("First Report");
		System.out.println(regularClass.data);
		regularClass.makeBackup();
		regularClass.setData("Second Report");
		System.out.println(regularClass.data);
		regularClass.recoverBackup();
		System.out.println(regularClass.data);
	}
	public class Memento{
		public String memoryData;

		public Memento(String data){
			this.memoryData=data;
		}
```

```
public String recoverOldInformation(){
                        return memoryData;
                }
        }
        public class RegularClass{
                Memento memento;
                String data;
                public RegularClass(){
                }
                public void setData(String data){
                        this.data = data;
                }
                public void makeBackup(){
                         memento = new Memento(data);
                }
                public void recoverBackup(){
                        data = memento.recoverOldInformation();
                }
        }
}
```

NULL OBJECT

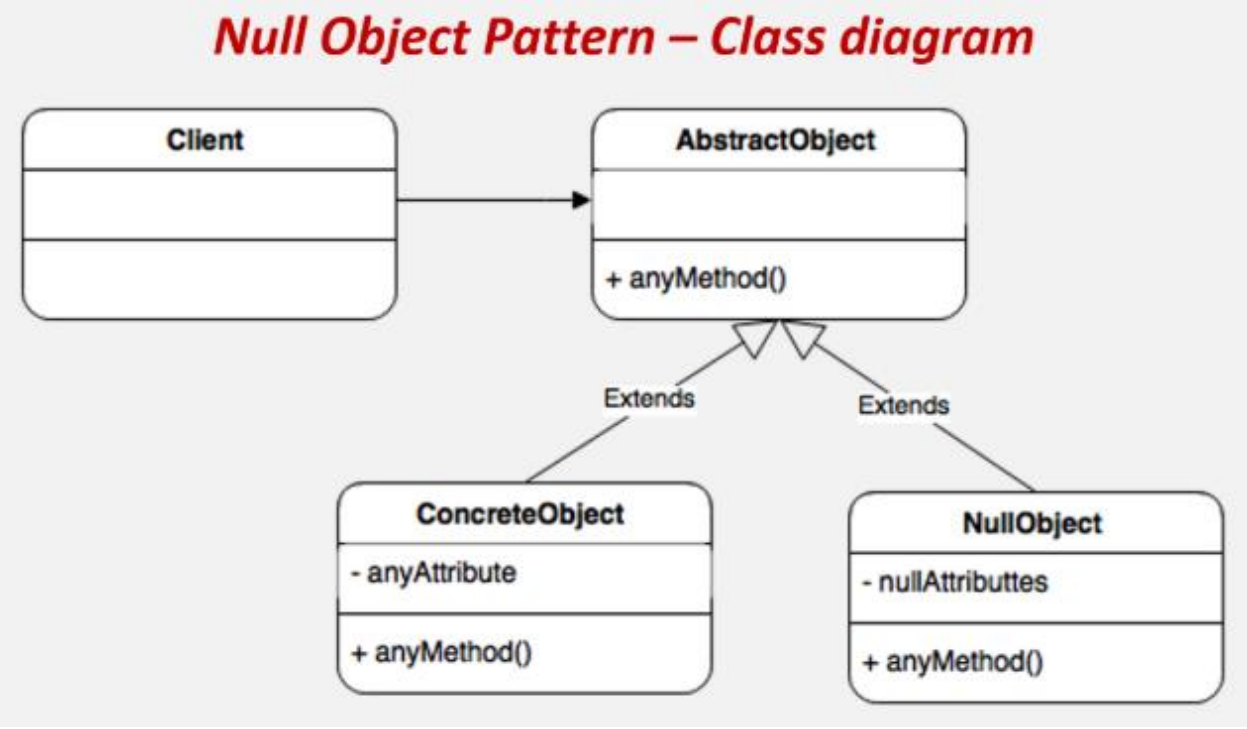
¿Qué es?

El patrón de diseño Null Object nace de la necesidad de evitar los valores nulos que puedan originar error en tiempo de ejecución. Básicamente lo que este patrón propone es utilizar instancias que implementen la interface requerida pero con un cuerpo vacío en lugar de regresar un valor null.

¿CUANDO USARLO?

- Una clase delega una operación a otra clase. La clase que delega normalmente no presta atención sobre como la otra clase implementa la operación. Sin embargo, algunas veces se requiere que la operación sea implementada sin hacer nada.
- Si quieres que la clase que delega la operación la delegue en todos los casos, incluyendo el caso de no hacer nada.
- Un objeto requiera de un colaborador. El patrón Null object no introduce esta colaboración; hace que se use una colaboración que ya existía.
- Alguna de las instancias del colaborador no haga nada.
- Se desee abstraer el manejo del "null" fuera del cliente.

Estructura



Implementación

```
/* Null Object Pattern implementation:*/
using System;
// Interfaz de los animales es la clave para la compatibilidad
// para las implementaciones de los animales inferiores.
interface IAnimal
{
   void Sonido();
}
// Perro es un animal real
class Perro : IAnimal
{
   public void Sonido()
   {
      Console.WriteLine("Guau!");
   }
}
// El caso nulo: esta clase NullAnimal deberá ser
   instanciada
// y utilizarse en lugar de la palabra clave null.
class NullAnimal : IAnimal
{
   public void Sonido()
   {
      // no se especifica ( a proposito)
      // ningún comportamiento..
   }
}
```

```
/* ========================
 * Ejemplo de uso simple de entrada principal.
 */
static class Program
{
   static void Main()
   {
      IAnimal dog = new Perro();
      dog.Sonido(); // Salida "Guau!"
     /* En lugar de utilizar C # Null, utilice una instancia
        NullAnimal.
 * Este ejemplo es simplista pero transmite la idea de
   que si
* una instancia de NullAnimal se utiliza entonces el programa
 * Nunca va a experimentar una System.NullReferenceException.
   NET
* en tiempo de ejecución.
 */
      IAnimal desconocido = new NullAnimal();
      //<< Reemplaza: IAnimal unknown = null;
      desconocido.Sonido();
     // Ninguna salida, pero no se producirá una excepción
        en tiempo de ejecución
   }
}
```

OBSERVER

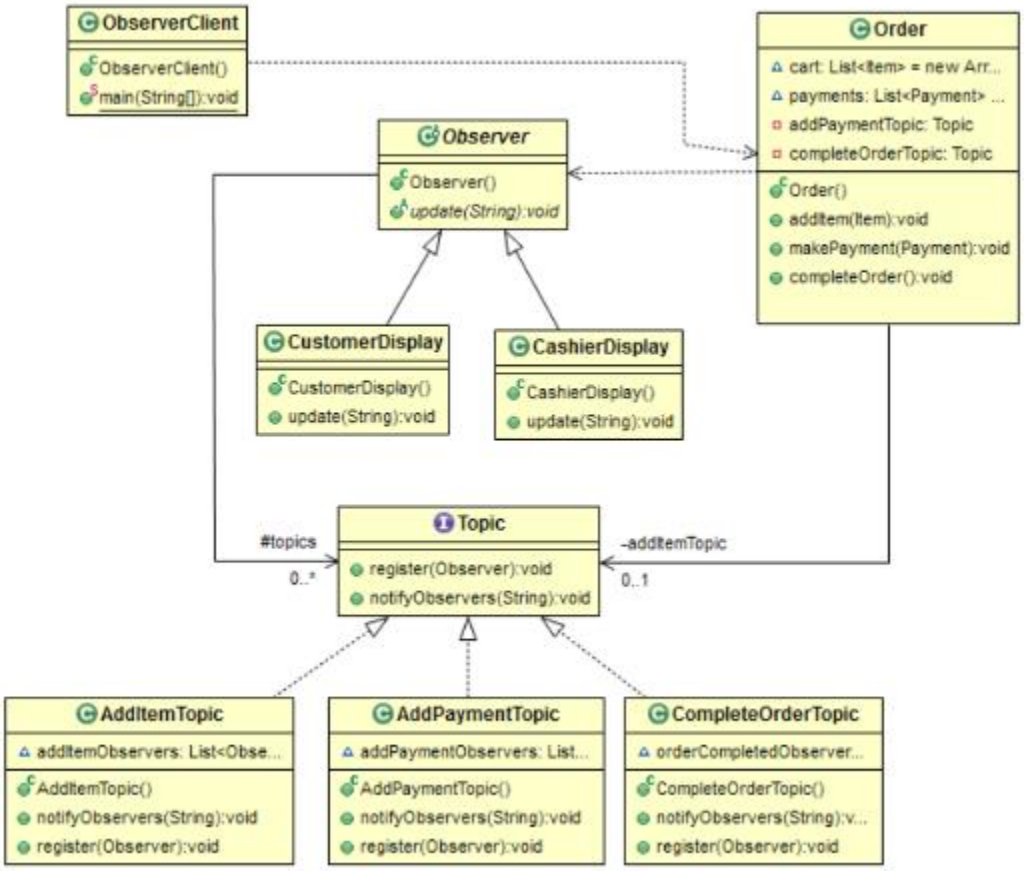
Cuándo usar

- Cuando hay una relación de uno a muchos entre objetos, como si uno el objeto se modifica, sus objetos dependientes deben ser notificados automáticamente y los cambios correspondientes se realizan a todos los objetos dependientes.

Intención

Definir una dependencia de uno a muchos entre objetos para que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente.

Estructura



Ejemplo en JAVA

En el caso de Java, el método de actualización seria notifyObservers.

```
import java.util.Scanner;
import java.util.Observable;
public class FuenteEvento extends Observable implements Runnable {
   public void run() {
      while (true) {
         String respuesta = new Scanner(System.in).next();
         setChanged();
         notifyObservers(respuesta);
      }
   }
}
import java.util.Observable;
public class MiApp {
   public static void main(String[] args) {
      System.out.println("Introducir Texto: ");
      FuenteEvento fuenteEvento = new FuenteEvento();
      fuenteEvento.addObserver(new Observer(){
            @Override
            public void update(Observable obj, Object arg){
               System.out.println("\nRespuesta recibida: " + arg);
            }
      });
      new Thread(fuenteEvento).start();
   }
}
```

DISEÑO DE SOFTWARE – CONCURRENCIA

ACTIVE OBJECT

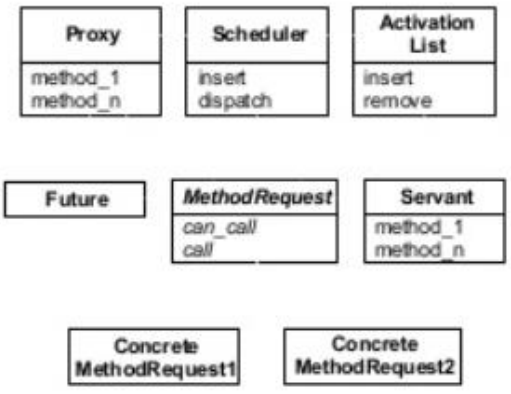
¿Qué es?

El patrón de diseño de objeto activo desacopla la ejecución del método de la invocación del método para los objetos que residen en su propio hilo de control. El objetivo es introducir la concurrencia, mediante el uso de la invocación de métodos asincrónicos y un programador para manejar solicitudes.

El patrón consta de seis elementos:

- Un proxy, que proporciona una interfaz hacia los clientes con métodos de acceso público.
- Una interfaz que define la solicitud del método en un objeto activo.
- Una lista de solicitudes pendientes de clientes.
- Un planificador, que decide qué solicitud ejecutar a continuación.
- La implementación del método de objeto activo.
- Una devolución de llamada o variable para que el cliente reciba el resultado.

Modelo



- A proxy provides an interface that allows clients to access methods of an object
- A concrete method request is created for every method invoked on the proxy
- A scheduler receives the method requests & dispatches them on the servant when they become runnable
- An activation list maintains pending method requests
- A servant implements the methods
- A future allows clients to access the results of a method call on the proxy

Implementación

Java

An example of active object pattern in Java.

Firstly we can see a standard class that provides two methods that set a double to be a certain value. This class does NOT conform to the active object pattern.

```
class MyClass {
   private double val = 0.0;
   void doSomething() {
      val = 1.0;
   }
   void doSomethingElse() {
      val = 2.0;
   }
}
```

The class is dangerous in a multithreading scenario because both methods can be called simultaneously, so the value of val (which is not atomic—it's updated in multiple steps) could be undefined—a classic race condition. You can, of course, use synchronization to solve this problem, which in this trivial case is easy. But once the class becomes realistically complex, synchronization can become very difficult.

To rewrite this class as an active object you could do the following:

```
class MyActiveObject {
   private double val = 0.0;
   private BlockingQueue<Runnable> dispatchQueue = new LinkedBlockingQueue<Runnable>();
   public MyActiveObject() {
      new Thread (new Runnable() {
            @Override
            public void run() {
               while (true) {
                  try {
                     dispatchQueue.take().run();
                  } catch (InterruptedException e) {
```

```
                    // okay, just terminate the dispatcher
                }
            }
        }
    }
    ).start();
  }
  void doSomething() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        @Override
        public void run() {
          val = 1.0;
        }
      }
    );
  }
  void doSomethingElse() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
        @Override
        public void run() {
          val = 2.0;
        }
      }
    );
  }
}
```

BINDING PROPIERTIES

¿Qué es?

El patrón de propiedades de enlace combina múltiples observadores para forzar que las propiedades en diferentes objetos se sincronicen o coordinen de alguna manera. Este patrón fue descrito por primera vez como una técnica por Victor Porton. Este patrón viene bajo patrones de concurrencia.

Comparación con la implementación orientada a aspectos

Como alternativa a la implementación orientada a aspectos de las propiedades mutuas, se puede proponer un enlace de propiedad. En la biblioteca LibPropC ++ C ++ también se implementa.

Alguna debilidad en el LibPropC ++ (con enlace de propiedad):

- Su uso no es transparente, ya que requiere que se declaren los atributos de objeto necesarios como propiedades y se deben proporcionar métodos de acceso adecuados
- El enlace de atributos en LibPropC ++ no está diseñado para reemplazar llamadas a métodos
- La biblioteca no mantiene un historial de interacción.

Modelo

Implementación

Code sketch for one-way binding may look like as follows:

```
bind_multiple_one_way(src_obj, src_prop, dst_objs[], dst_props[])
{
 for (i, j) in (dst_objs, dst_props)
 {
  bind_properties_one_way(src_obj, src_prop, i, j);
 }
}
```

Two-way binding can be expressed as follows (in C++):

```
// In this pseudo-code are not taken into the account initial values assignments
bind_two_way(prop1, prop2)
{
  bind(prop1, prop2);
  bind(prop2, prop1);
}
```

Accomplishing the binding (i.e. connecting the property change notification in an event handler) may be like as follows:

```
on_property_change(src_prop, dst_prop)
{
  block_signal(src_obj, on_property_change);
  dst_prop := src_prop;
  unblock_signal(src_obj, on_property_change);
}
```

BALKING

¿Qué es?

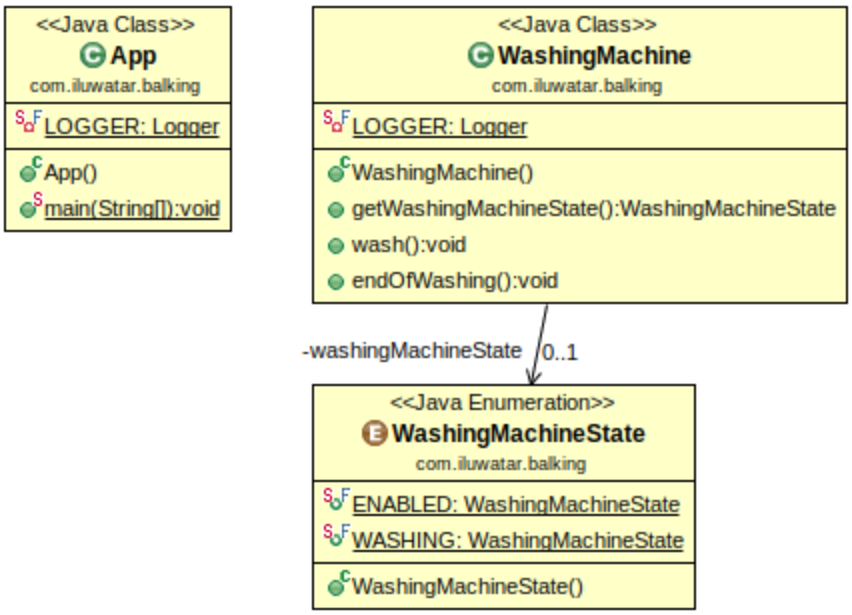
Es una propiedad de los sistemas en la cual los procesos de un cómputo se hacen simultáneamente, y pueden interactuar entre ellos.

Por ejemplo, un objeto lee un archivo y proporciona métodos para acceder al contenido del archivo. Cuando el archivo no está abierto y se intenta llamar a un método para acceder al contenido, el objeto “rechaza” la petición.

Los objetos que utilizan este patrón son generalmente sólo en un estado que es propenso a balking temporalmente, pero para una cantidad desconocida de tiempo.

Balking se utiliza con un único patrón de ejecución roscado para ayudar a coordinar el cambio de un objeto en el estado.

Modelo



Implementación

```
public class Example {
   private boolean jobInProgress = false;
   public void job() {
      synchronized(this) {
         if (jobInProgress) {
            return;
         }
         jobInProgress = true;
      }
      // Code to execute job goes here
      // ...
      jobCompleted();
   }
   void jobCompleted() {
      synchronized(this) {
         jobInProgress = false;
      }}}
```

MONITOR

¿Qué es?

El patrón gira en torno a la sincronización. En resumen, los hilos concurrentes (clientes) solo pueden usar el objeto a través de un conjunto de métodos sincronizados. Solo se puede ejecutar un método a la vez. Por lo general, un método sincronizado observa una determinada condición. Sin embargo, no hay encuestas involucradas. En cambio, los métodos están siendo notificados. Esa es una diferencia importante en comparación con el objeto activo.

Monitor Object es similar al Active Object en el sentido de que expone una interfaz definida a través de los métodos sincronizados del objeto. Por otro lado, los métodos se ejecutan en el hilo del cliente ya que no existe la noción de un control de hilo centralizado. Tampoco hay una sobrecarga de rendimiento significativa, ya que los bucles ineficientes de espera ocupada (sondeo) se reemplazan con notificaciones.

Componentes clave

Supervisar objeto: expone los métodos sincronizados como el único medio de acceso del cliente

Métodos sincronizados: garantice un acceso seguro para subprocesos al estado interno

Monitor de bloqueo: utilizado por los métodos sincronizados para serializar la invocación de métodos

Condición del monitor: abastece la programación de ejecución cooperativa

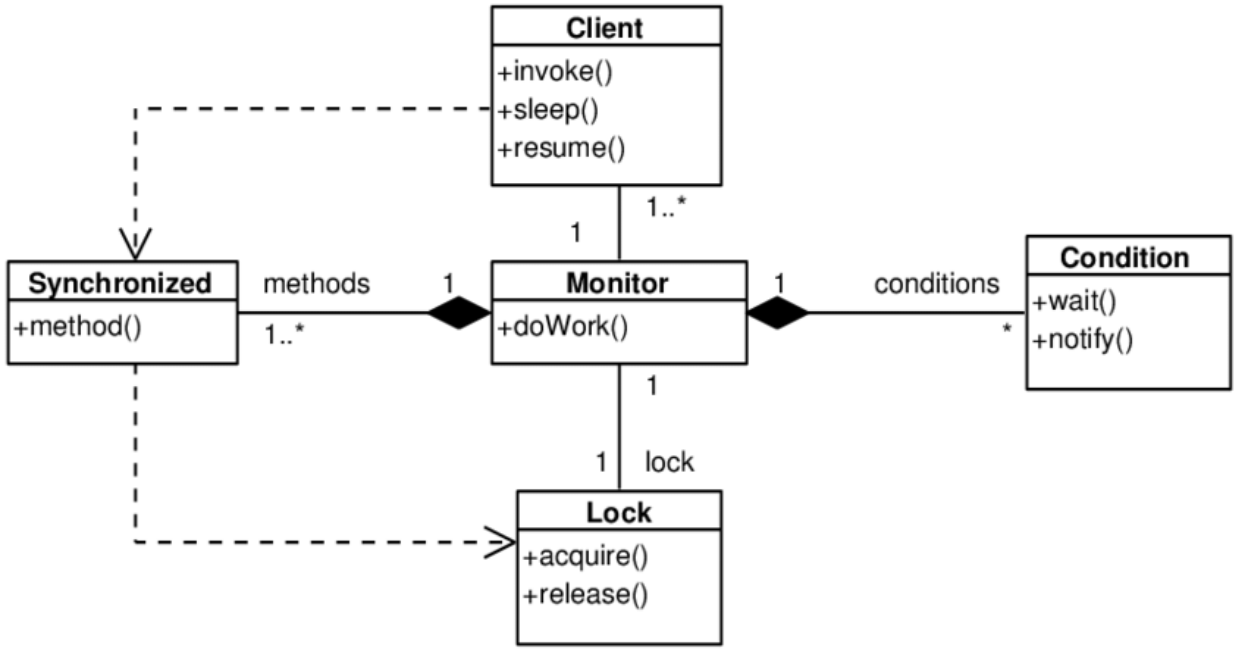
Ventajas

Sincronización simplificada: todo el trabajo duro se descarga en el objeto mismo, los clientes no están preocupados por los problemas de sincronización.

Programación de ejecución cooperativa: las condiciones del monitor se utilizan para suspender / reanudar la ejecución del método.

Reducción de la sobrecarga de rendimiento: notificaciones sobre sondeos ineficientes.

Modelo




```
public class ThreadSafeClass {
        private Object lock1 = new Object();
        private Object lock2 = new Object();
        public void methodA() {
                synchronized(lock1) {
                        //critical section A
                }
        }
        public void methodB() {
                synchronized(lock1) {
                        //critical section B
                }
        }
        public void methodC() {
                synchronized(lock2) {
                        //critical section C
```

```
            }
        }
        public void methodD() {
            synchronized(lock2) {
                //critical section D
            }
        }
}
```

REACTOR

¿Qué es?

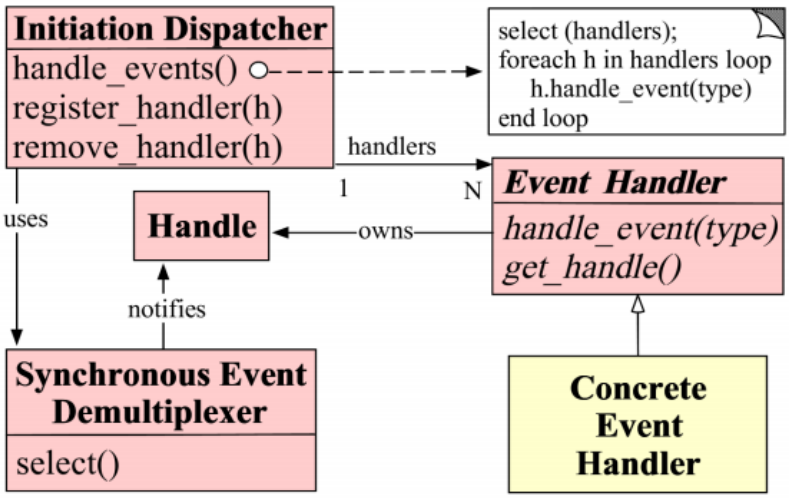
El patrón de diseño reactor es un patrón de programación concurrente para manejar los pedidos de servicio entregados de forma concurrente a un manejador de servicio desde una o más entradas. El manejador de servicio demultiplexa los pedidos entrantes y los entrega de forma sincrónica a los manejadores de pedidos asociados.

Recursos: cualquier medio que puede proveer entrada o salida del sistema.

Demultiplexor síncrono de eventos: utiliza un bucle de eventos para bloquear a todos los recursos. Cuando es posible comenzar una operación síncrona en un recurso sin necesidad de bloqueo, el demultiplexor lo envía al despachador.

Despachador: maneja el registro y la baja de los manejadores de pedidos. Entrega los recursos desde el multiplexor al manejador de pedidos asociado

Estructura



Implementación

```
resources = [socketA, socketB, pipeA];
while(!resources.isEmpty()) {
  for(i = 0; i < resources.length; i++) {
    resource = resources[i];
    //try to read
    var data = resource.read();
    if(data === NO_DATA_AVAILABLE)
      //there is no data to read at the moment
      continue;
    if(data === RESOURCE_CLOSED)
      //the resource was closed, remove it from the list
      resources.remove(i);
    else
      //some data was received, process it
      consumeData(data);
  }
}
```

Event demultiplexing

```
socketA, pipeB;
watchedList.add(socketA, FOR_READ);       //[1]
watchedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(watchedList)) {    //[2]
  //event loop
  foreach(event in events) {         //[3]
    //This read will never block and will always return data
    data = event.resource.read();
    if(data === RESOURCE_CLOSED)
      //the resource was closed, remove it from the watched list
```

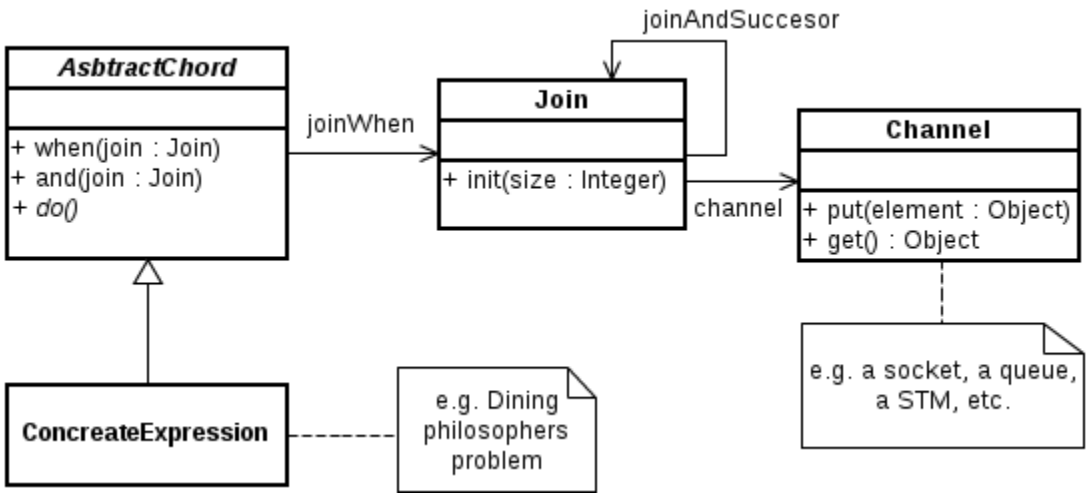
```
    demultiplexer.unwatch(event.resource);
  else
    //some actual data was received, process it
    consumeData(data);
 }
}
```

JOIN

¿Qué es?

Join-patterns proporciona una forma de escribir programas informáticos concurrentes , paralelos y distribuidos mediante el paso de mensajes . En comparación con el uso de subprocesos y bloqueos, este es un modelo de programación de alto nivel que utiliza el modelo de construcciones de comunicación para abstraer la complejidad del entorno concurrente y permitir la escalabilidad . Se centra en la ejecución de un acorde entre mensajes atómicamente consumidos de un grupo de canales.

Estructura



Implementación

"Join-patterns can be used to easily encode related concurrency idioms like actors and active objects."

Barriers

```
class SymmetricBarrier {
public readonly Synchronous.Channel Arrive;
public SymmetricBarrier(int n) {
   // create j and init channels (elided)
   var pat = j.When(Arrive);
   for (int i = 1; i < n; i++) pat = pat.And(Arrive);
   pat.Do(() => { });
}
}
```

Dining philosophers problem

```
var j = Join.Create();
Synchronous.Channel[] hungry;
Asynchronous.Channel[] chopstick;
j.Init(out hungry, n); j.Init(out chopstick, n);
for (int i = 0; i < n; i++) {
   var left = chopstick[i];
   var right = chopstick[(i+1) % n];
   j.When(hungry[i]).And(left).And(right).Do(() => {
   eat(); left(); right(); // replace chopsticks
   });
}
```

Mutual exclusion

```
class Lock {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Lock() {
```

```
      // create j and init channels (elided)
      j.When(Acquire).And(Release).Do(() => { });
      Release(); // initially free
    }
}
```

Producers/Consumers

```
class Buffer<T> {
    public readonly Asynchronous.Channel<T> Put;
    public readonly Synchronous<T>.Channel Get;
    public Buffer() {
        Join j = Join.Create(); // allocate a Join object
        j.Init(out Put);
        // bind its channels
        j.Init(out Get);
        j.When(Get).And(Put).Do // register chord
        (t => { return t; });
    }
}
```

Reader-writer locking

```
class ReaderWriterLock {
    private readonly Asynchronous.Channel idle;
    private readonly Asynchronous.Channel<int> shared;
    public readonly Synchronous.Channel AcqR, AcqW, RelR, RelW;
    public ReaderWriterLock() {
    // create j and init channels (elided)
    j.When(AcqR).And(idle).Do(() => shared(1));
    j.When(AcqR).And(shared).Do(n => shared(n+1));
    j.When(RelR).And(shared).Do(n => {
    if (n == 1) idle(); else shared(n-1);
```

```
    });
    j.When(AcqW).And(idle).Do(() => { });
    j.When(RelW).Do(() => idle());
    idle(); // initially free
}
}
```

Semaphores

```
class Semaphore {
    public readonly Synchronous.Channel Acquire;
    public readonly Asynchronous.Channel Release;
    public Semaphore(int n) {
        // create j and init channels (elided)
        j.When(Acquire).And(Release).Do(() => { });
        for (; n > 0; n--) Release(); // initially n free
    }
}
```

DOUBLE CHECK

¿Qué es?

Es un patrón de diseño de software utilizado para reducir la sobrecarga de adquirir un bloqueo al probar el criterio de bloqueo (la "pista de bloqueo") antes adquiriendo la cerradura. El bloqueo ocurre solo si la verificación del criterio de bloqueo indica que se requiere un bloqueo.

El patrón, cuando se implementa en algunas combinaciones de idioma / hardware, puede ser inseguro. A veces, puede considerarse un antipatrón .

Uso

Por lo general, se usa para reducir la sobrecarga de bloqueo al implementar la " inicialización diferida " en un entorno de subprocesos múltiples, especialmente como parte del patrón Singleton . La inicialización diferida evita inicializar un valor hasta la primera vez que se accede a él.

Implementación

Para empezar, consideremos un singleton simple con sincronización draconiana:

```
public class DraconianSingleton {
   private static DraconianSingleton instance;
   public static synchronized DraconianSingleton getInstance() {
      if (instance == null) {
         instance = new DraconianSingleton();
      }
      return instance;
   }

   // private constructor and other methods ...
}
```

A pesar de que esta clase es segura para subprocesos, podemos ver que hay un claro inconveniente en el rendimiento: cada vez que queremos obtener la instancia de nuestro singleton, necesitamos adquirir un bloqueo potencialmente innecesario.

Para solucionarlo, podríamos comenzar verificando si necesitamos crear el objeto en primer lugar y solo en ese caso obtendríamos el bloqueo.

Yendo más allá, queremos realizar la misma verificación nuevamente tan pronto como ingresemos al bloque sincronizado, para mantener la operación atómica:

```
public class DclSingleton {
   private static volatile DclSingleton instance;
   public static DclSingleton getInstance() {
      if (instance == null) {
         synchronized (DclSingleton .class) {
            if (instance == null) {
               instance = new DclSingleton();
      }
      return instance;
 // private constructor and other methods...
}
```

PATRONES DE ARQUITECTURA

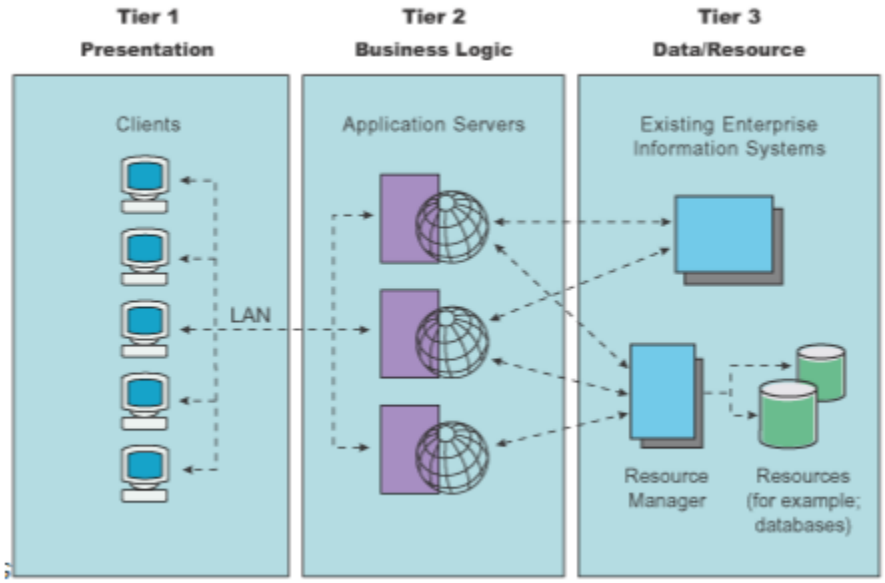
3 niveles.

También conocida como arquitectura de tres capas, la arquitectura de tres capas define cómo organizar el modelo de diseño en capas, que pueden estar físicamente distribuidas, lo cual quiere decir que los componentes de una capa sólo pueden hacer referencia a componentes en capas inmediatamente inferiores.

Uso.

Simplificación la comprensión y la organización del desarrollo de sistemas complejos, reduciendo las dependencias de forma que las capas más bajas no son conscientes de ningún detalle o interfaz de las superiores. Además, nos ayuda a identificar qué puede reutilizarse, y proporciona una estructura que nos ayuda a tomar decisiones sobre qué partes comprar y qué partes construir.

Diseño.



Por Capas

se enfoca en la distribución de roles y responsabilidades de forma jerárquica proveyendo una forma muy efectiva de separación de responsabilidades. El rol indica el modo y tipo de interacción con otras capas, y la responsabilidad indica la funcionalidad que está siendo desarrollada.

Uso.

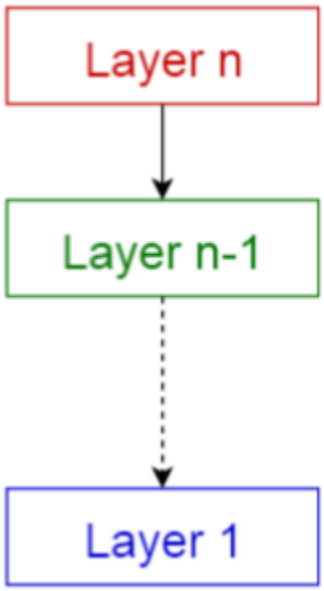
• Describe la descomposición de servicios de forma que la mayoría de la interacción ocurre solamente entre capas vecinas.

• Las capas de una aplicación pueden residir en la misma maquina física (misma capa) o puede estar distribuido sobre diferentes computadores (n-capas).

• Los componentes de cada capa se comunican con otros componentes en otras capas a través de interfaces muy bien definidas.

• Este modelo ha sido descrito como una “pirámide invertida de reúso” donde cada capa agrega responsabilidad y abstracción a la capa directamente sobre ella.

Diseño.



SOA

es el nexo que une las metas de negocio con el sistema de software. Su papel es el de aportar flexibilidad, desde la automatización de las infraestructura y herramientas necesarias consiguiendo, al mismo tiempo, reducir los costes de integración.

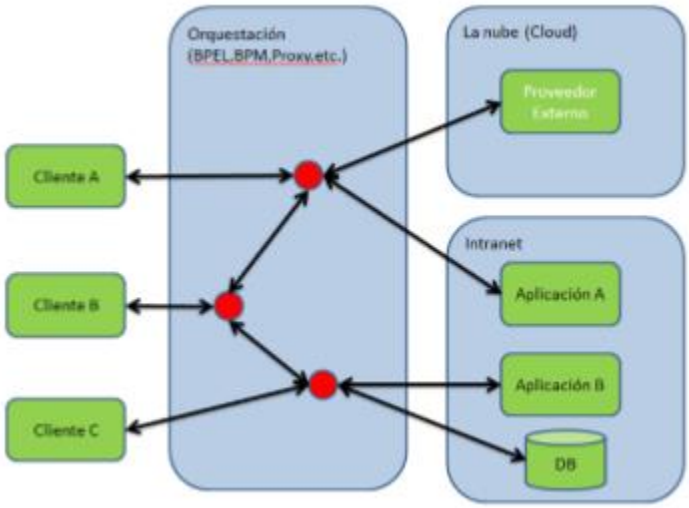
SOA se ocupa del diseño y desarrollo de sistemas distribuidos y es un potente aliado a la hora de llevar a cabo la gestión de grandes volúmenes de datos, datos en la nube y jerarquías de datos.

Uso.

permite la reutilización de activos existentes para nuevos servicios que se pueden crear a partir de una infraestructura de TI que ya se había diseñado. De esta forma, permite a las empresas optimizar la

inversión por medio de la reutilización que, además, conlleva otra ventaja: la interoperabilidad entre las aplicaciones y tecnologías heterogéneas.

Diseño.



Microservicios

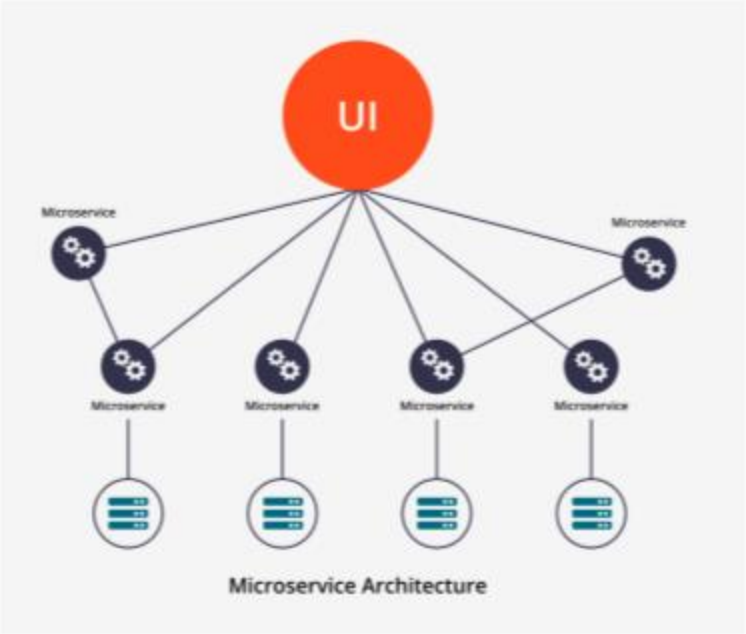
son tanto un estilo de arquitectura como un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus elementos más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los microservicios son

elementos independientes que funcionan en conjunto para llevar a cabo las mismas tareas. Cada uno de esos elementos o procesos es un microservicio.

Uso.

distribuir sistemas de software de calidad con mayor rapidez, lo cual es posible gracias a los microservicios, pero también se deben considerar otros aspectos. Dividir las aplicaciones en microservicios no es suficiente; es necesario administrarlos, coordinarlos y gestionar los datos que crean y modifican.

Diseño.



MVC

es un patrón de diseño arquitectónico de software, que sirve para clasificar la información, la lógica del sistema y la interfaz que se le presenta al usuario. En este tipo de arquitectura existe un sistema central o controlador que gestiona las entradas y la salida del sistema, uno o varios modelos que se encargan de buscar los datos e información necesaria y una interfaz que muestra los resultados al usuario final.

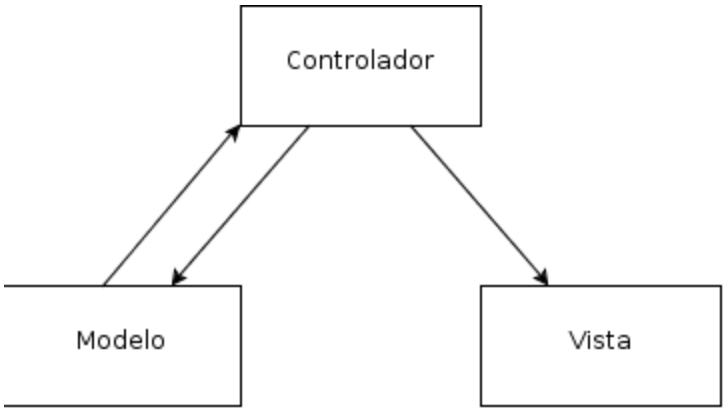
Uso.

- Modelo: este componente se encarga de manipular, gestionar y actualizar los datos. Si se utiliza una base de datos aquí es donde se realizan las consultas, búsquedas, filtros y actualizaciones.

- Vista: este componente se encarga de mostrarle al usuario final las pantallas, ventanas, páginas y formularios; el resultado de una solicitud. Desde la perspectiva del programador este componente es el que se encarga del frontend; la programación de la interfaz de usuario si se trata de una aplicación de escritorio, o bien, la visualización de las páginas web (CSS, HTML, HTML5 y JavaScript).

- Controlador: este componente se encarga de gestionar las instrucciones que se reciben, atenderlas y procesarlas. Por medio de él se comunican el modelo y la vista: solicitando los datos necesarios; manipulándolos para obtener los resultados; y entregándolos a la vista para que pueda mostrarlos.

Diseño.



P2P

son una red de computadoras que funciona sin necesidad de contar ni con clientes ni con servidores fijos, lo que le otorga una flexibilidad que de otro modo sería imposible de lograr. Esto se obtiene gracias a que la red trabaja en forma de una serie de nodos

que se comportan como iguales entre sí. Esto en pocas palabras significa que las computadoras conectadas a la red P2P actual al mismo tiempo como clientes y servidores con respecto a los demás computadores conectadas.

Uso.

Las redes P2P son muy útiles para todo lo que tiene que ver con compartir datos entre usuarios, y es muy utilizada en la actualidad para compartir entre los usuarios que se conecten con cualquiera de los clientes que existen en el mercado todo tipo de material, tanto de video, como de audio, programas y literatura, entre otros.

Diseño.

Imagen 1 – Red tradicional cliente/servidor

Imagen 2 – Red P2P

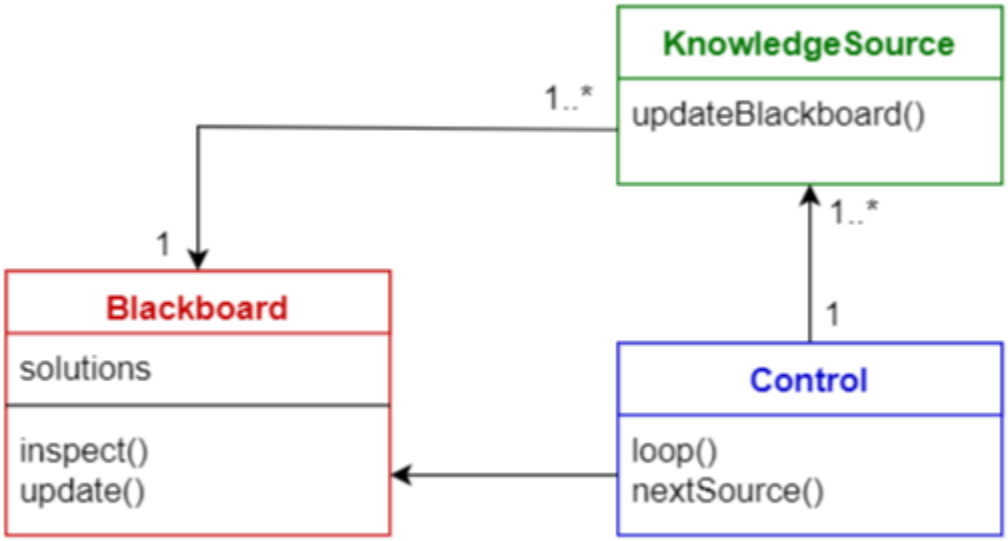
PIZARRA

Es un modelo arquitectónico de software habitualmente utilizado en sistemas expertos, multiagente y basados en el conocimiento. La arquitectura en pizarra blackboard consta de múltiples elementos funcionales, denominados agentes, y un instrumento de control denominado pizarra. Los agentes están especializados en resolver una tarea concreta. Todos ellos cooperan para alcanzar una meta común, si bien, sus objetivos individuales no están aparentemente coordinados.

Uso.

es tremendamente útil cuando el problema a resolver (o algoritmo a implementar) es extremadamente complejo en términos cognitivos. Es decir, cuando el flujo de control del algoritmo es enrevesado, o simplemente, no se tiene un conocimiento completo del problema a resolver.

Diseño.



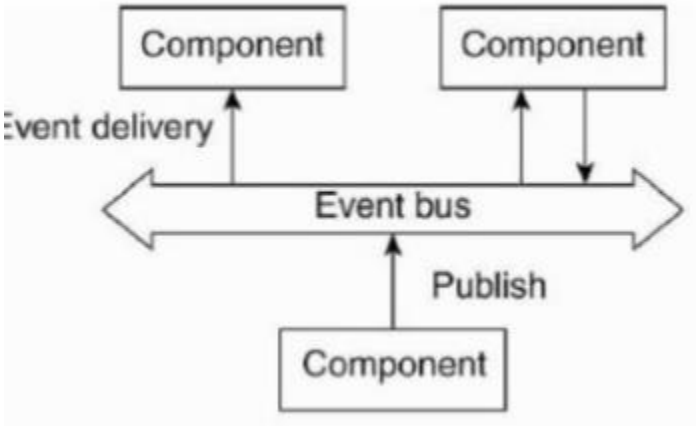
EVENTOS

es un modelo y una arquitectura de software que sirve para diseñar aplicaciones. En un sistema como este, la captura, la comunicación, el procesamiento y la persistencia de los eventos son la estructura central de la solución. Esto difiere del modelo tradicional basado en solicitudes. Los eventos son aquellos sucesos o cambios significativos en el estado del hardware o el software de un sistema. Un evento y su notificación no son lo mismo: la segunda es un mensaje que el sistema envía para comunicar a otra parte del sistema que se ha producido cierto evento.

Uso.

Esta arquitectura está compuesta por consumidores y productores de eventos. El productor detecta los eventos y los representa como mensajes. No conoce al consumidor del evento ni el resultado que generará este último. Después de la detección de un evento, este se transmite del productor a los consumidores a través de los canales de eventos, donde se procesan de manera asíncrona con una plataforma de procesamiento de eventos. Debe notificarse a los consumidores del evento sobre su ocurrencia, para que lo procesen o simplemente se vean afectados por él.

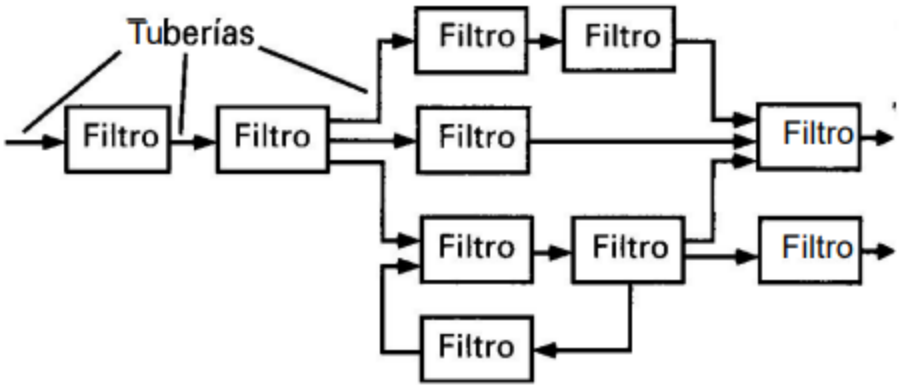
Diseño.



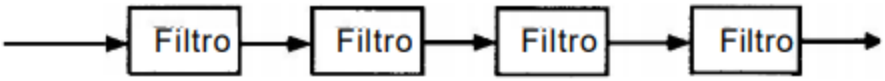
TUBERIAS Y FILTROS

Este patrón se puede usar para estructurar sistemas que producen y procesan una secuencia de datos. Cada paso de procesamiento se incluye dentro de un componente de filtro. Los datos que se procesarán se pasan a través de las tuberías. Estas tuberías se pueden utilizar para el almacenamiento en búfer o con fines de sincronización. Uso. - Compiladores Los filtros consecutivos realizan análisis léxico, análisis sintáctico y generación de código. - Flujos de trabajo en bioinformática.

Diseño.



(a) Tuberías y filtros



(b) Secuencial por lotes

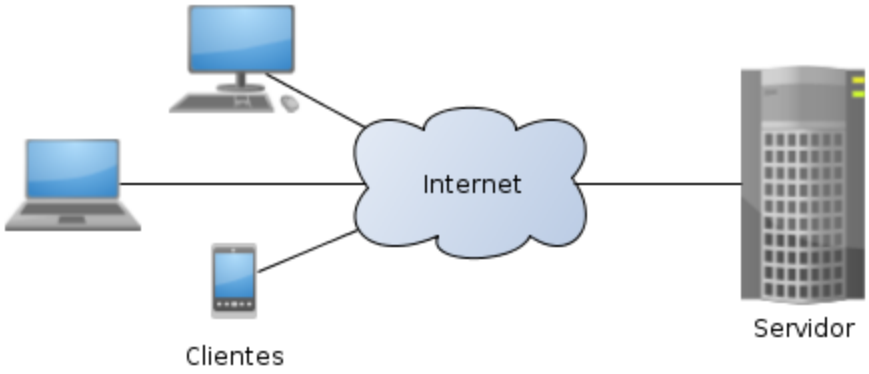
CLIENTE-SERVIDOR

Este patrón consiste en dos partes; un servidor y múltiples clientes. El componente del servidor proporcionará servicios a múltiples componentes del cliente. Los clientes solicitan servicios del servidor y el servidor proporciona servicios relevantes a esos clientes. Además, el servidor sigue escuchando las solicitudes de los clientes.

USO.

Aplicaciones en línea como correo electrónico, uso compartido de documentos y banca.

Diseño.



PUBLICADOR-SUSCRIPTOR

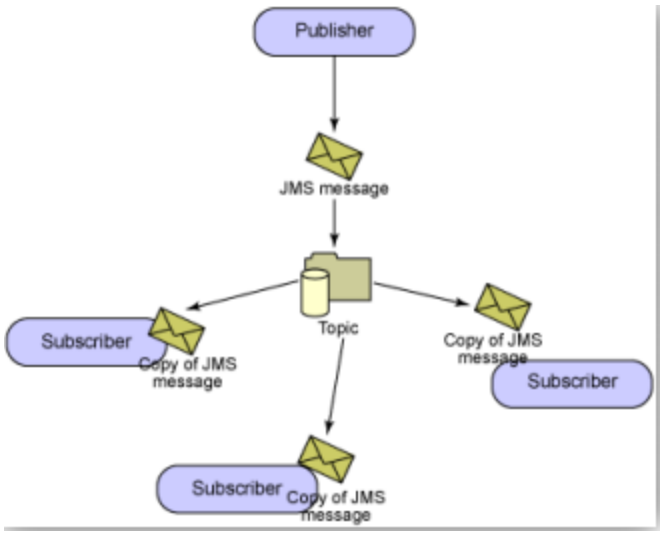
Permita que una aplicación anuncie eventos de forma asincrónica a varios consumidores interesados, sin necesidad de emparejar los remitentes con los receptores.

USO.

- Un canal de mensajería de entrada utilizado por el remitente. El remitente empaqueta los eventos en mensajes, mediante un formato de mensaje conocido, y envía estos mensajes a través del canal de entrada. El remitente en este patrón también se denomina publicador. (Un mensaje es un paquete de datos. Un evento es un mensaje que notifica a otros componentes sobre un cambio o una acción que ha tenido lugar).

- Un canal de mensajería de salida por consumidor. Los consumidores se conocen como suscriptores. - Un mecanismo para copiar cada mensaje del canal de entrada a los canales de salida para todos los suscriptores interesados en ese mensaje. Normalmente, esta operación se controla mediante un intermediario, como un bus de mensajes de agente o un evento.

Diseño.


Publisher
JMS message
Topic
Subscriber
Copy of JMS message
Copy of JMS message
Subscriber
Subscriber
Copy of JMS message