Tarea 1 (Patrones de diseño)

Arquitectura.

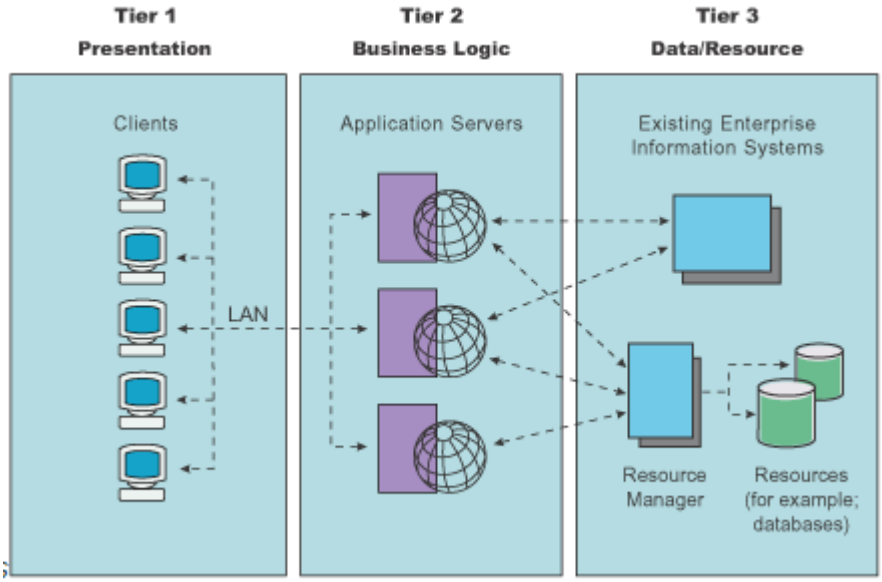
❖3 NIVELES.

También conocida como arquitectura de tres capas, la arquitectura de tres capas define cómo organi-zar 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.



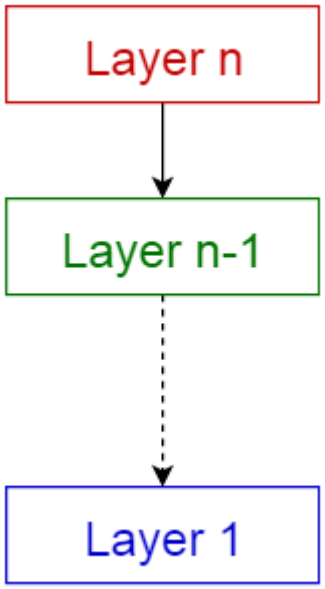
❖ Por Capas

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.



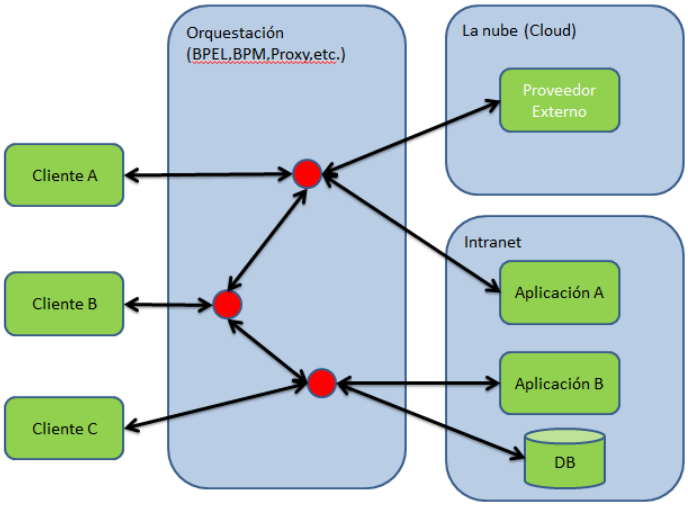
❖SOA

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

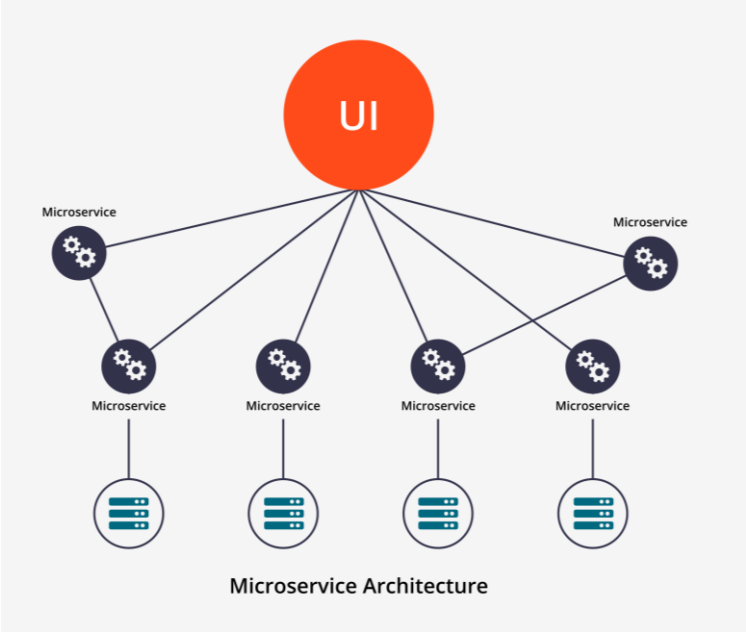
Son tanto un estilo de arquitectura como un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus elementos más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los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

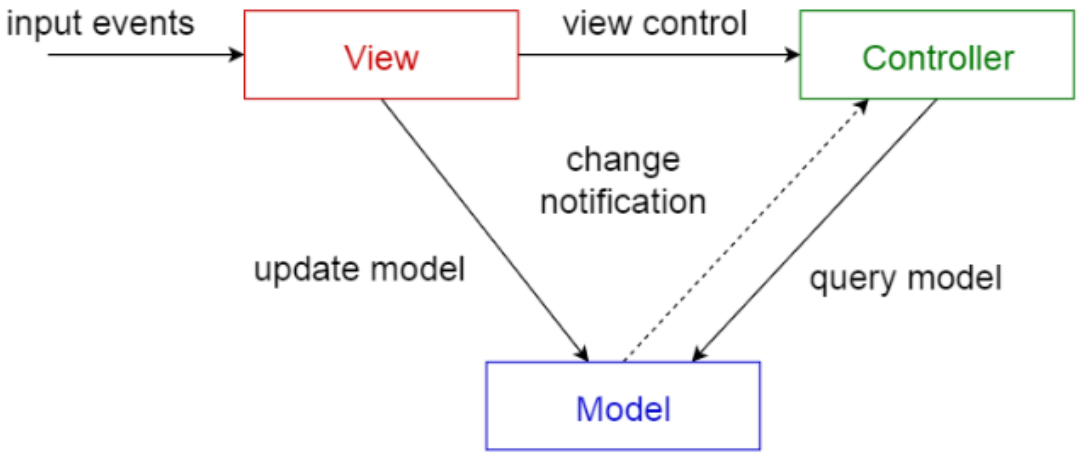
es un patrón de diseño arquitectónico de software, que sirve para clasificar la información, la lógica del sistema y la interfaz que se le presenta al usuario. En este tipo de arquitectura existe un sistema central o controlador que gestiona las entradas y la salida del sistema, uno o varios modelos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

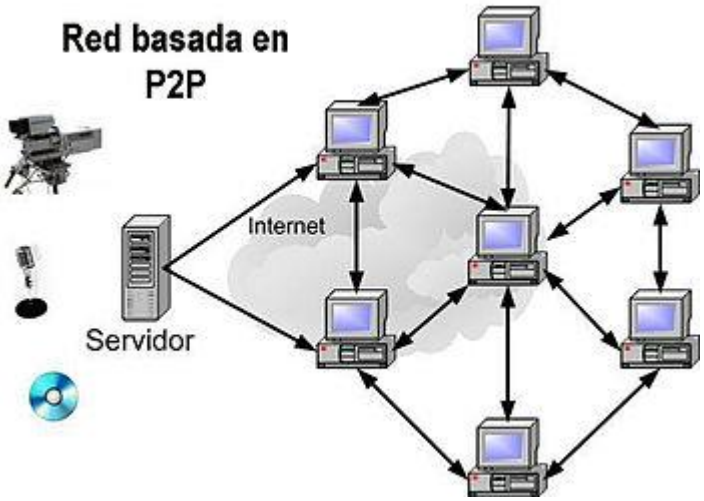
son una red de computadoras que funciona sin necesidad de contar ni con clientes ni con servidores fijos, lo que le otorga una flexibilidad que de otro modo sería imposible de lograr. Esto se obtiene gracias a que la red trabaja en forma de una serie de nodos

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.



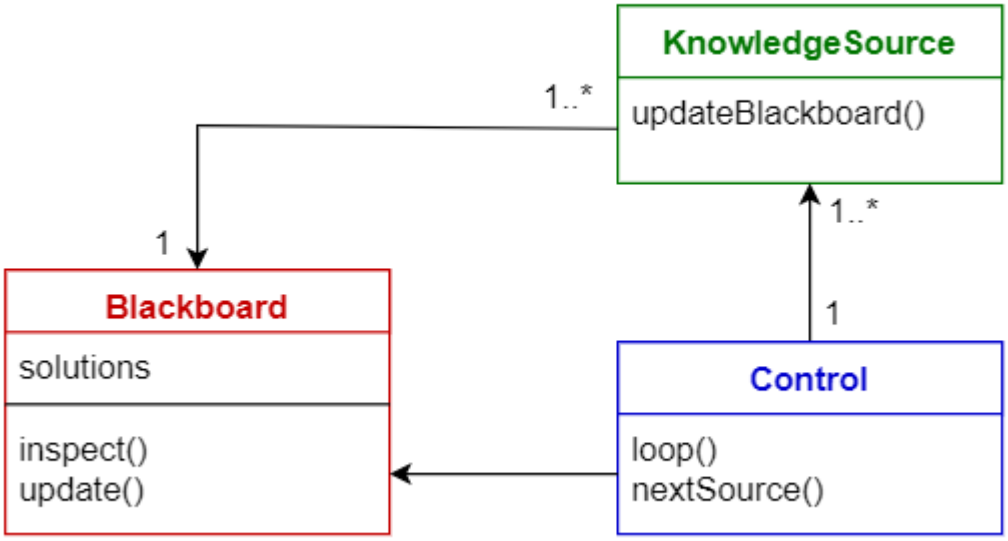
❖ PIZARRAA

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.

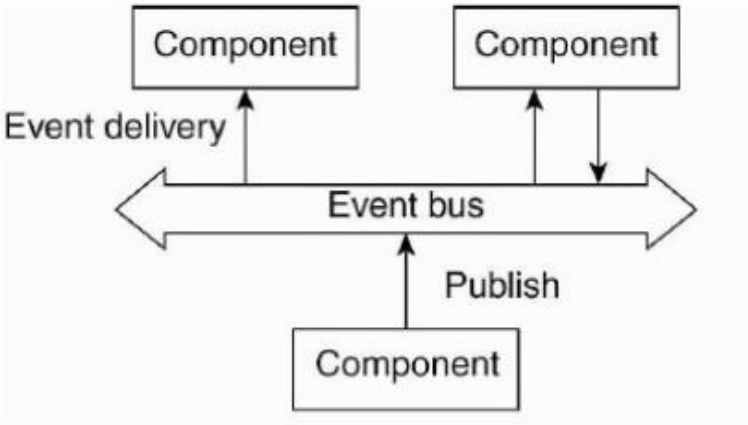
Los eventos son aquellos sucesos o cambios significativos en el estado del hardware o el software de un sistema. Un evento y su notificación no son lo mismo: la segunda es un mensaje que el sistema envía para comunicar a otra parte del sistema que se ha producido cierto evento.

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.



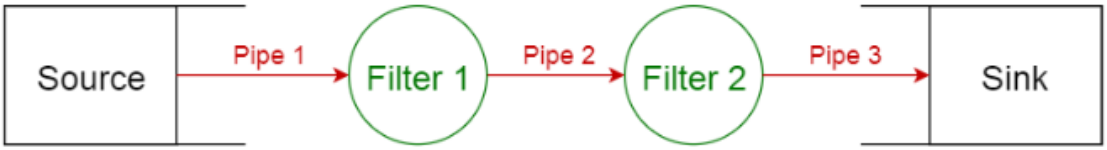
❖Tuberias y filtros

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.



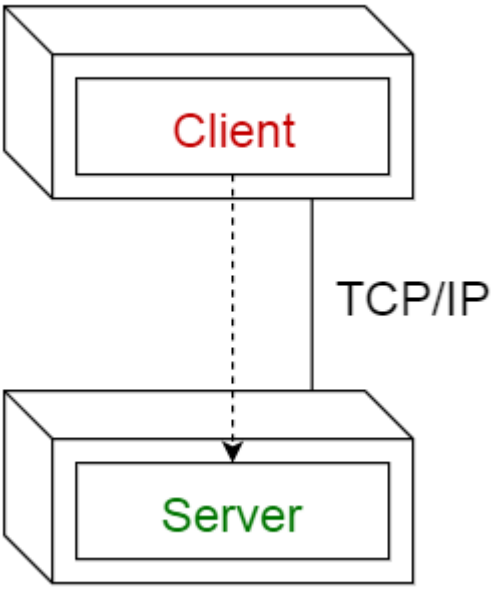
❖ Cliente-servidor

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.



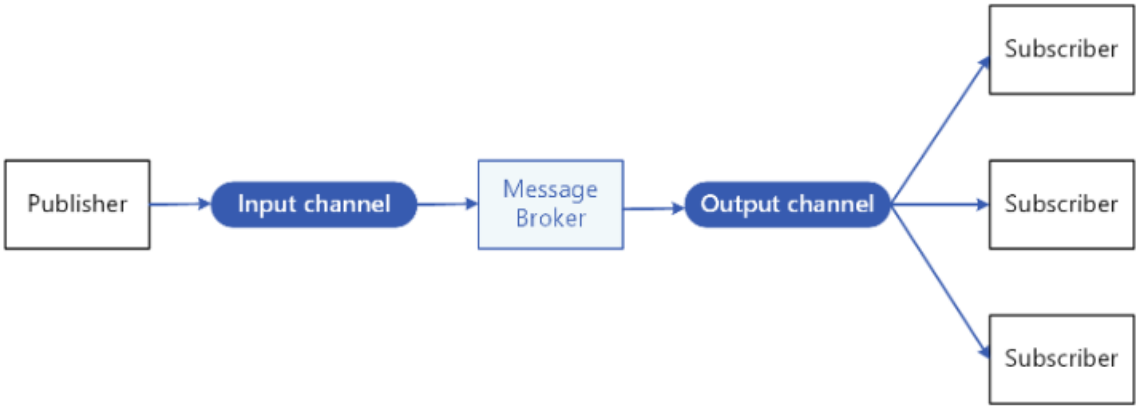
❖ PUBLICADOR-SERVIDOR

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.



Diseño de Software.

Creación.

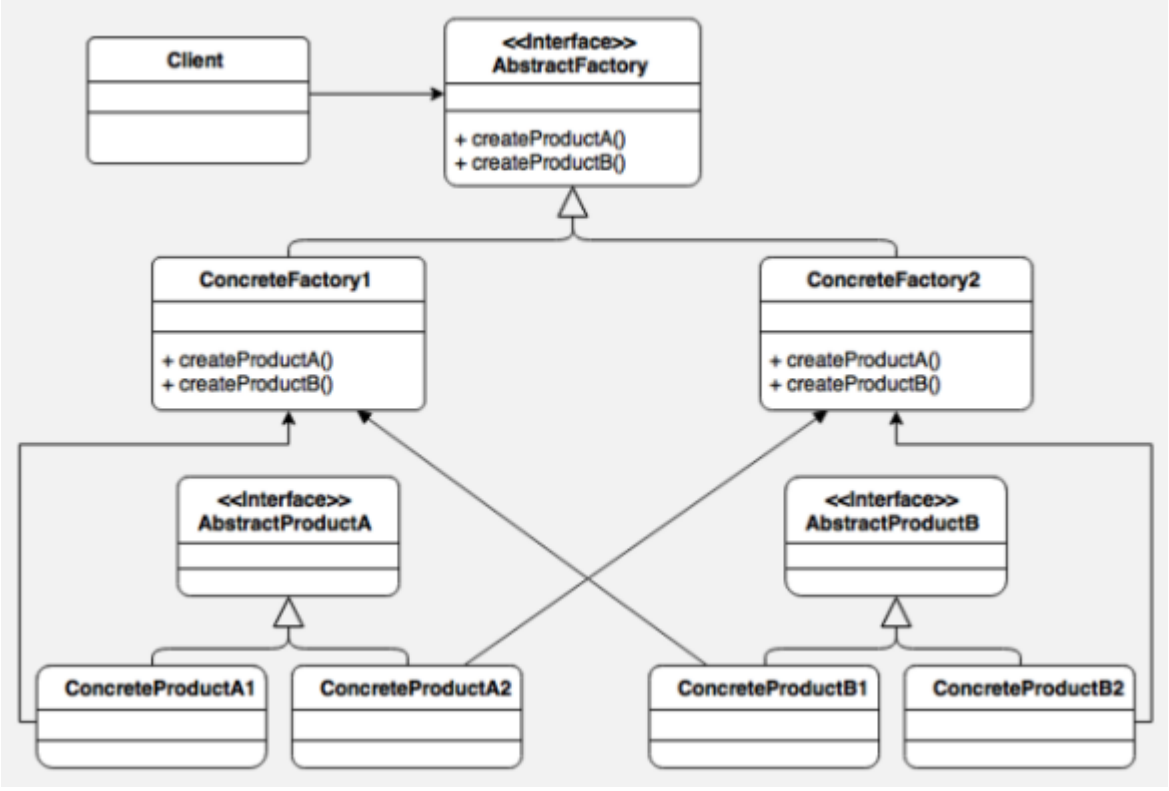
❖ABSTRACT FACTORY.

El patrón de diseño Abstract Factory busca agrupar un conjunto de clases que tiene un funcionamiento en común llamadas familias, las cuales son creadas mediante un Factory.

USO.

Este patrón es especialmente útil cuando requerimos tener ciertas familias de clases para resolver un problema, sin embargo, puede que se requieran crear implementaciones paralelas de estas clases para resolver el mismo problema, pero con una implementación distinta.

Diseño.



Código.

```
01.  package AbstractFactory;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          // Se insertarán los componentes con el borde azul
08.          Cliente cliente = new Cliente( new ComponentesAzules() );
09.
10.          System.out.println("----------------------------");
11.
12.          // Ahora se insertarán los componentes con el borde rojo
13.          cliente = new Cliente( new ComponentesRojos() );
14.      }
15.  }
```

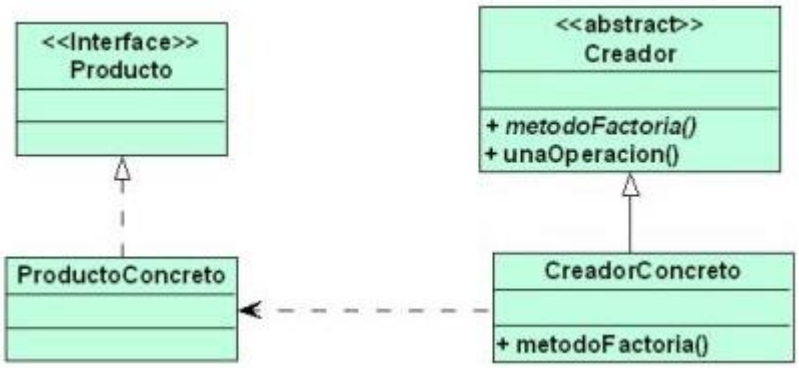
❖FACTORY METHOD

Expone un método de creación, delegando en las subclases la implementación de este método.

USO.

Podemos utilizar este patrón cuando definamos una clase a partir de la que se crearán objetos, pero sin saber de qué tipo son, siendo otras subclases las encargadas de decidirlo.

Diseño.



Código.

```
01.  package FactoryMethod1;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          CreadorAbstracto creator = new Creador();
08.
09.          IArchivo audio = creator.crear( Creador.AUDIO );
10.          audio.reproducir();
11.
12.          IArchivo video = creator.crear( Creador.VIDEO );
13.          video.reproducir();
14.      }
15.  }
```

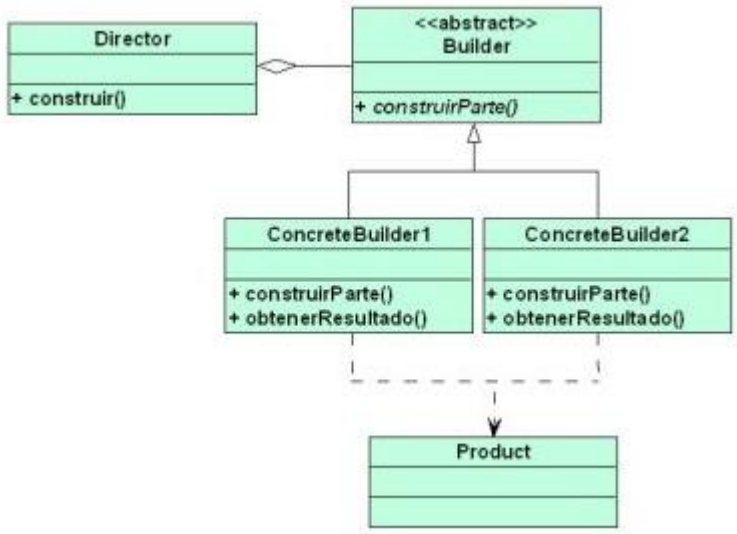
❖BUILDER

Separa la creación de un objeto complejo de su estructura, de tal forma que el mismo proceso de construcción nos puede servir para crear representaciones diferentes.

USO.

Este patrón puede ser utilizado cuando necesitemos crear objetos complejos compuestos de varias partes independientes.

Diseño.



Código.

```
22.
23.             // Construir un coche con equipamiento full
24.             objFabrica.construir( full );
25.             Coche cocheFull = full.getCoche();
26.
27.             // Mostrar la información de cada coche creado
28.             mostrarCaracteristicas( cocheBase );
29.             mostrarCaracteristicas( cocheMedio );
30.             mostrarCaracteristicas( cocheFull );
31.         }
32.
33.         // --------------------------------
34.
35.         public static void mostrarCaracteristicas( Coche coche )
36.         {
37.             System.out.println( "Motor: " + coche.getMotor() );
38.             System.out.println( "Carrocería: " + coche.getCarroceria() );
39.             System.out.println( "Elevalunas eléctrico: " + coche.getElevalunasElec() );
40.             System.out.println( "Airea acondicionado: " + coche.getAireAcond() );
41.
42.             System.out.println("==================================");
43.         }
44.     }
```

❖SINGLENTON

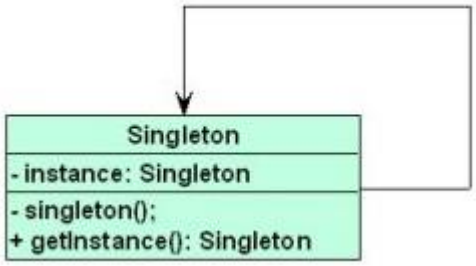
Limita a uno el número de instancias posibles de una clase en nuestro programa, y proporciona un acceso global al mismo.

Uso.

Utilizaremos el patrón Singleton cuando por alguna razón necesitemos que exista sólo una instancia (un objeto) de una determinada Clase.

Dicha clase se creará de forma que tenga una propiedad estática y un constructor privado, así como un método público estático que será el encargado de crear la instancia (cuando no exista) y guardar una referencia a la misma en la propiedad estática (devolviendo ésta) .

Diseño.



CODIGO.

text pop-up

```
01.  package Singleton;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          for(int num=1; num<=5; num++)
08.          {
09.            Coche.getInstancia();
10.          }
11.      }
12.  }
```

```
01.  package Singleton;
02.
03.  public class Coche
04.  {
05.      private static Coche instancia;
06.
07.       // ----------------------------------------
08.
09.       private Coche() {
10.       }
11.
12.       // ----------------------------------------
13.
14.      public static Coche getInstancia()
15.       {
16.          if (instancia == null) {
17.              instancia = new Coche();
18.               System.out.println("El objeto ha sido creado");
19.           }
20.           else {
21.               System.out.println("Ya existe el objeto");
22.           }
23.
24.          return instancia;
25.      }
26.  }
```

❖ PROTOTYPE

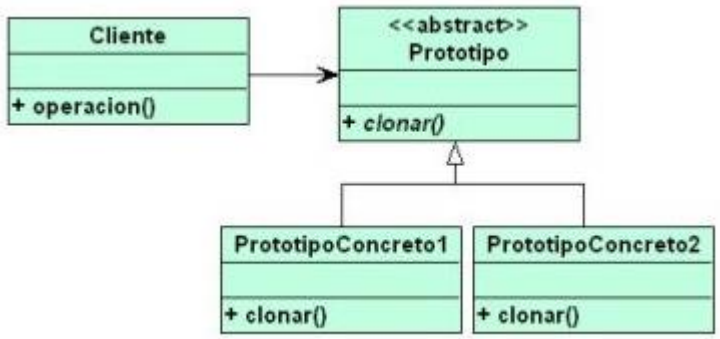
Permite la creación de objetos basados en «plantillas». Un nuevo objeto se crea a partir de la clonación de otro objeto.

USO.

Este patrón nos será útil si necesitamos crear y manipular copias de otros objetos. Cuando nos disponemos a clonar un objeto es importante tener en cuenta si guarda referencias de otros o no. En este punto hay que distinguir las siguientes formas de replicarlos:

- **Copia superficial:** el objeto clonado tendrá los mismos valores que el original, guardando también referencias a otros objetos que contenga (por lo que si son modificados desde el objeto original o desde alguno de sus clones el cambio afectará a todos ellos) .
- **Copia profunda:** el objeto clonado tendrá los mismos valores que el original así como copias de los objetos que contenga el original (por lo que si son modificados por cualquiera de ellos, el resto no se verán afectados) .

DISEÑO.



CODIGO.

```
29.
30.             // Modificamos el Guerrero clonado
31.             g2.setNombre("Warrior2");
32.             g2.setArma("HACHA");
33.
34.             // Mostrar los datos de ambos Guerreros (ambos tienen diferentes datos)
35.             System.out.println("Tras modificar el clon, ahora se llama [" + g2.getNombre() + "]");
36.             System.out.println("Su arma es [" + g2.getArma() + "]\n");
37.
38.             System.out.println("El nombre del Guerrero original es [" + g1.getNombre() + "]");
39.             System.out.println("Su arma es [" + g1.getArma() + "]");
40.
41.             System.out.println("============================");
42.
43.             // Modificamos el Guerrero original
44.             g1.setNombre("Warrior-1");
45.             g1.setArma("MAZA");
46.
47.             // Mostrar los datos de ambos Guerreros tras la modificación (ambos tienen diferentes datos)
48.             System.out.println("Tras modificar el original, ahora es [" + g1.getNombre() + "]");
49.             System.out.println("Su arma es [" + g1.getArma() + "]\n");
50.
51.             System.out.println("El nombre del clon es [" + g2.getNombre() + "]");
52.             System.out.println("Su arma es [" + g2.getArma() + "]");
53.         }
54.     }
```

❖DEPENDENCY INJECTION

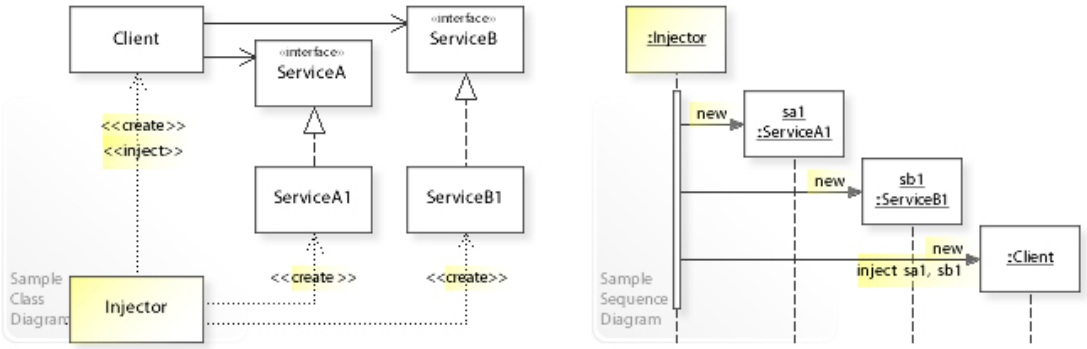
ES UNA TÉCNICA MEDIANTE LA CUAL UN OBJETO SUMINISTRA LAS DEPENDENCIAS DE OTRO OBJETO. UNA "DEPENDENCIA" ES UN OBJETO QUE SE PUEDE UTILIZAR, POR EJEMPLO, COMO UN SERVICIO . EN LUGAR DE QUE UN CLIENTE ESPECIFIQUE QUÉ SERVICIO USARÁ, ALGO LE DICE AL CLIENTE QUÉ SERVICIO USAR. LA "INYECCIÓN" SE REFIERE AL PASO DE UNA DEPENDENCIA (UN SERVICIO) AL OBJETO (UN CLIENTE) QUE LO USARÍA. EL SERVICIO FORMA PARTE DEL ESTADO DEL CLIENTE . [1] PASAR EL SERVICIO AL CLIENTE, EN LUGAR DE PERMITIR QUE UN CLIENTE CONSTRUYA O ENCUENTRE EL SERVICIO, ES EL REQUISITO FUNDAMENTAL DEL PATRÓN.

USO.

- LA INYECCIÓN DE DEPENDENCIA LE PERMITE AL CLIENTE LA FLEXIBILIDAD DE SER CONFIGURABLE. SOLO EL COMPORTAMIENTO DEL CLIENTE ES FIJO. EL CLIENTE PUEDE ACTUAR SOBRE CUALQUIER COSA QUE ADMITA LA INTERFAZ INTRÍNSECA QUE EL CLIENTE ESPERA.

- La inyección de dependencia se puede utilizar para externalizar los detalles de configuración de un sistema en archivos de configuración, permitiendo que el sistema se reconfigure sin re-compilación. Se pueden escribir configuraciones separadas para diferentes situaciones que requieren diferentes implementaciones de componentes. Esto incluye, pero no se limita a, pruebas.

Diseño.



Codigo.

```
// Un ejemplo sin inyección de dependencia
public class Client {
    // Referencia interna al servicio utilizado por este cliente
    privado ExampleService service ;

    // Constructor
    Client () {
        // Especifica una implementación específica en el constructor en lugar de utilizar el
        servicio de inyección de dependencias = new ExampleService ();
    }

    // Método dentro de este cliente que utiliza los servicios
    public String greet () {
        return "Hello" + service . getName ();
    }
}
```

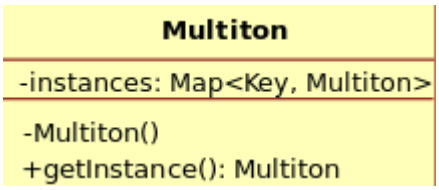
❖ Multiton

El patrón multiton es un patrón de diseño similar al Singleton, que permite que solo se cree una instancia de una clase. El patrón multiton se expande en el concepto singleton para administrar un mapa de instancias nombradas como pares clave-valor.

Uso.

El patrón multiton permite la creación controlada de múltiples instancias, que administra mediante el uso de un mapa. En lugar de tener una sola instancia por aplicación (por ejemplo, el `java.lang.Runtime` en el lenguaje de programación Java) , el patrón multitono garantiza una sola instancia por clave .

Diseño.



Codigo.

```
usando  System.Collections.Generic ;

public  enum  MultitonType  {
    Zero ,
    One ,
    Two
};

pública  clase  Multiton  {
    privado  estática  de sólo lectura  IDictionary < MultitonType ,  Multiton >  casos  =
        nuevo  diccionario < MultitonType ,  Multiton > ();
    privados int  número ;

    Multiton privado  ( número int ) { esto . número = número ; } pública estática Multiton GetInstance ( MultitonType tipo ) {
// Lazy init (no hilo de seguridad como por escrito) // Recomendar el uso de Double Check bloqueo si necesitara hilo de seguridad si
(! Instancias . ContainsKey ( tipo ))
        {
            instancias . Add ( type ,  new  Multiton (( int ) type ));
        }
        instancias de retorno  [ tipo ]; } cadena de anulación pública ToString () { devuelve "Mi número es" + número .
ToString (); } // Ejemplo de uso público estático vacío

    Main ( string []  args )  {
        Multiton  m0  =  Multiton . GetInstance ( MultitonType . Zero );
        Multitón  m1  =  Multitón . GetInstance ( MultitonType . Uno );
        Multiton  m2  =  Multitón . GetInstance ( MultitonType . Dos );
        Sistema . Consola . WriteLine (m0 );
        Sistema . Consola . WriteLine ( m1 );
        Sistema . Consola . WriteLine ( m2 );
    }
}
```

Estructura.

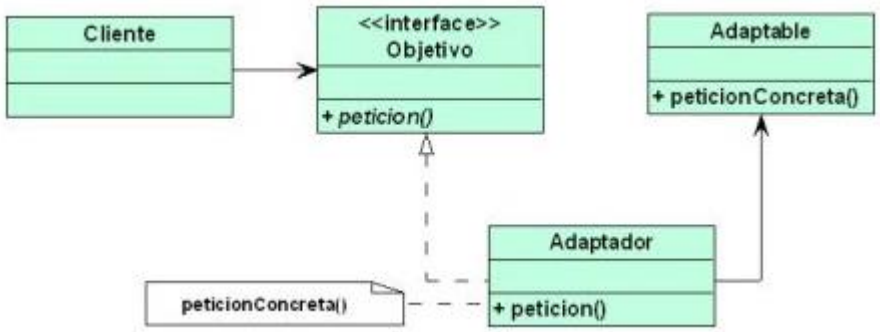
❖Adapter

Permite a dos clases con diferentes interfaces trabajar entre ellas, a través de un objeto intermedio con el que se comunican e interactúan.

Uso.

Este patrón permite que trabajen juntas clases con interfaces incompatibles.
Para ello, un objeto adaptador reenvía al otro objeto los datos que recibe (a través de los métodos que implementa, definidos en una clase abstracta o interface) tras manipularlos en caso necesario.

Diseño.



Codigo.

```
01.  package estructurales.adapter.adapter01;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          Adaptador conversor = new Adaptador();
08.
09.           conversor.ingresarPesetas( 2000 );
10.           conversor.ingresarPesetas( 5000 );
11.           conversor.ingresarPesetas( 1000 );
12.
13.           System.out.println( "Total euros: " + conversor.getSaldo() );
14.      }
15.  }
```

❖ Bridge

Desacopla una abstracción de su implementación, para que las dos puedan evolucionar de forma independiente.

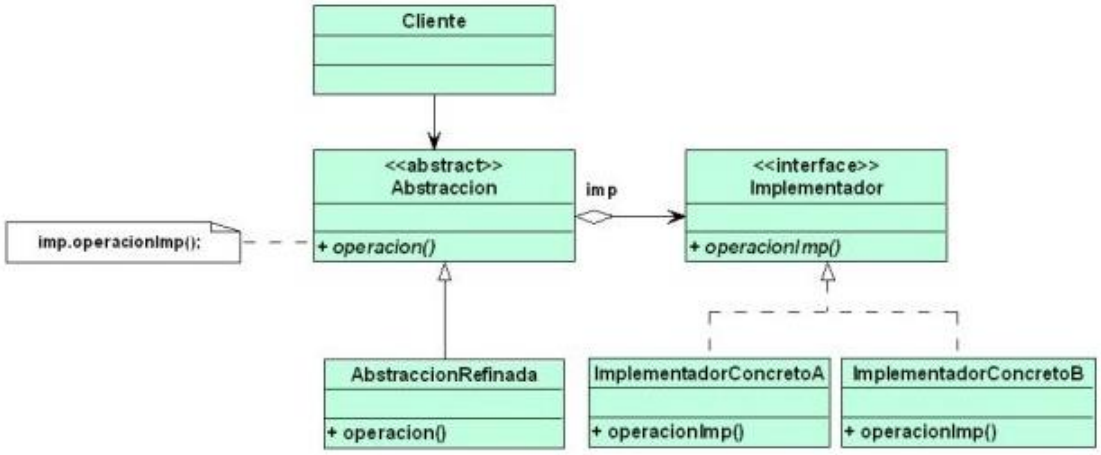
Uso.

Según el libro de GoF este patrón de diseño permite desacoplar una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

Supongamos que tenemos una clase abstracta en la que se define un método que deberá implementar cada clase que herede de ella: ¿cómo haríamos si una clase hija necesitase implementarlo de forma que realizase acciones diferentes dependiendo de determinadas circunstancias?

En dichos casos nos resultaría útil el patrón Bridge (puente) ya que 'desacopla una abstracción' (un método abstracto) al permitir indicar (durante la ejecución del programa) a una clase qué 'implementación' del mismo debe utilizar (qué acciones ha de realizar) .

DISEÑO.



CODIGO.

text pop-up

```
01.  package estructurales.bridge.bridge01;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          // Crear un objeto de tipo 'AbstraccionRefinada' indicándole un 'ImplementadorConcreto'
08.          ElaborarAlimento lasagna = new ElaborarLasagna( new Carne() );
09.          lasagna.obtener();
10.
11.          // Ahora le indicamos que use otra implementación para obtener la de verduras
12.          lasagna.setImplementador( new Verduras() );
13.          lasagna.obtener();
14.      }
15.  }
```

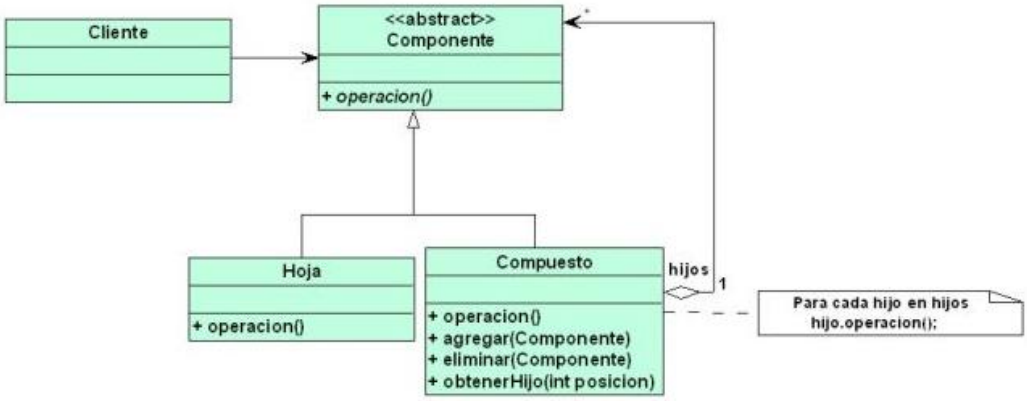
❖COMPOSITE

Facilita la creación de estructuras de objetos en árbol, donde todos los elementos emplean una misma interfaz. Cada uno de ellos puede a su vez contener un listado de esos objetos, o ser el último de esa rama.

USO.

Este útil patrón permite crear y manejar estructuras de objetos en forma de árbol, en las que un objeto puede contener a otro(s) .

DISEÑO.



Las estructuras de este tipo se componen de nodos (un objeto que a su vez contiene otros objetos) y Hojas (objetos que no contienen otros) , y que ambos comparten una misma Interface que define métodos que deben implementar.

CODIGO.

```java
package estructurales.composite.composite01;

public class Main
{
    public static void main(String[] args)
    {
        // Crear la carpeta principal e insertar archivos
        Carpeta c1 = new Carpeta("CARPETA_1");
        c1.insertarNodo( new Archivo("Archivo1.txt") );
        c1.insertarNodo( new Archivo("Archivo2.txt") );
        c1.insertarNodo( new Archivo("Archivo3.txt") );

        // Crear una subcarpeta e insertar archivos
        Carpeta c2 = new Carpeta("CARPETA_2");
        c2.insertarNodo( new Archivo("Archivo4.txt") );
        c2.insertarNodo( new Archivo("Archivo5.txt") );

        // Insertar la subcarpeta dentro de la principal
        c1.insertarNodo( c2 );

        // Insertar otro archivo dentro de la carpeta principal
        c1.insertarNodo( new Archivo("Archivo6.txt") );

        c1.mostrar();

        System.out.println("--------------------");

        // Eliminamos la subcarpeta (junto con su contenido)
        c1.eliminarNodo( c2 );

        c1.mostrar();
    }
}
```

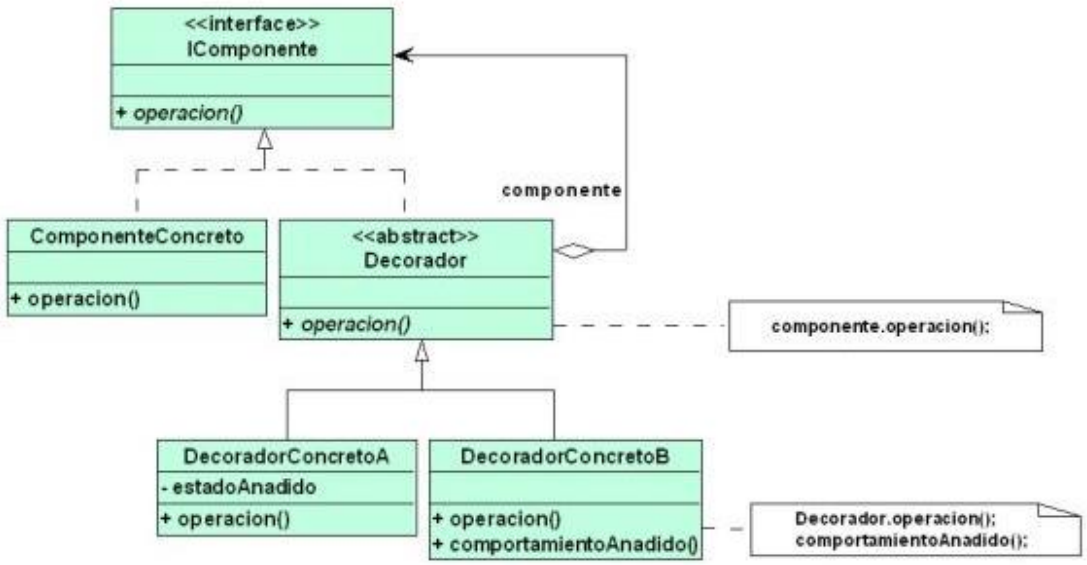
❖ DECODATOR

Permite añadir funcionalidad extra a un objeto (de forma dinámica o estática) sin modificar el comportamiento del resto de objetos del mismo tipo.

USO.

Sencillo e interesante patrón que permite añadir funcionalidades a un objeto en aquellos casos en los que no sea necesario o recomendable hacerlo mediante herencia.

DISEÑO.



CODIGO.

```
01.  package estructurales.decorator.decorator01;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          IVentana ventana1 = new Ventana();
08.          ventana1.dibujar(50, 70);
09.
10.          System.out.println("-----");
11.
12.          // Dibujar una ventan que tenga la barra de desplazamiento vertical
13.          IVentana ventana2 = new DecoradorDesplazamientoVert( new Ventana() );
14.          ventana2.dibujar(300, 200);
15.
16.          System.out.println("-----");
17.
18.          // Dibujar una ventan que tenga la barra de desplazamiento horizontal
19.          IVentana ventana3 = new DecoradorDesplazamientoHoriz( new Ventana() );
20.          ventana3.dibujar(400, 300);
21.
22.          System.out.println("-----");
23.
24.          // Dibujar una ventan que tenga las barras de desplazamientos horizontal y vertical
25.          IVentana ventana4 = new DecoradorDesplazamientoVert( new DecoradorDesplazamientoHoriz( new Ventana(
26.          ventana4.dibujar(100, 120);
27.      }
28.  }
```

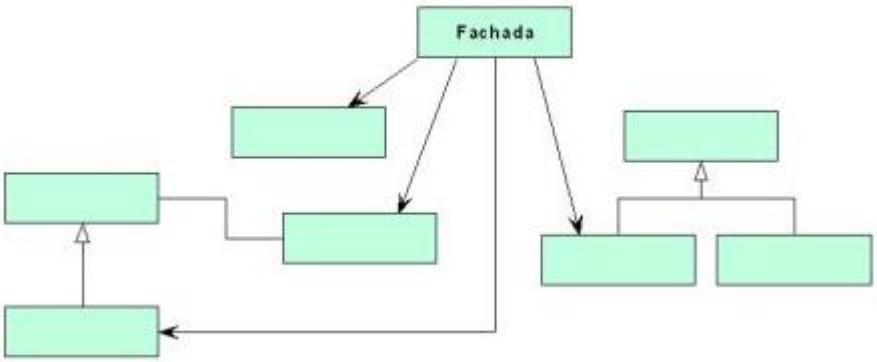
❖ FACADE

Una facade (o fachada) es un objeto que crea una interfaz simplificada para tratar con otra parte del código más compleja, de tal forma que simplifica y aísla su uso. Un ejemplo podría ser crear una fachada para tratar con una clase de una librería externa.

USO.

Este patrón nos permite acceder a un subsistema de forma más sencilla.

DISEÑO.



CODIGO.

```
01.  package estructurales.facade.facade01;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          ComprobarLiquidos liquidos = new ComprobarLiquidos();
08.          liquidos.comprobar();
09.
10.          ComprobarAsiento asiento = new ComprobarAsiento();
11.          asiento.comprobar();
12.
13.          ComprobarEspejos espejos = new ComprobarEspejos();
14.          espejos.comprobar();
15.
16.          Arrancar arrancar = new Arrancar();
17.          arrancar.encenderContacto();
18.
19.          System.out.println("\nProceso finalizado.");
20.      }
21.  }
```

❖ FLYWEIGHT

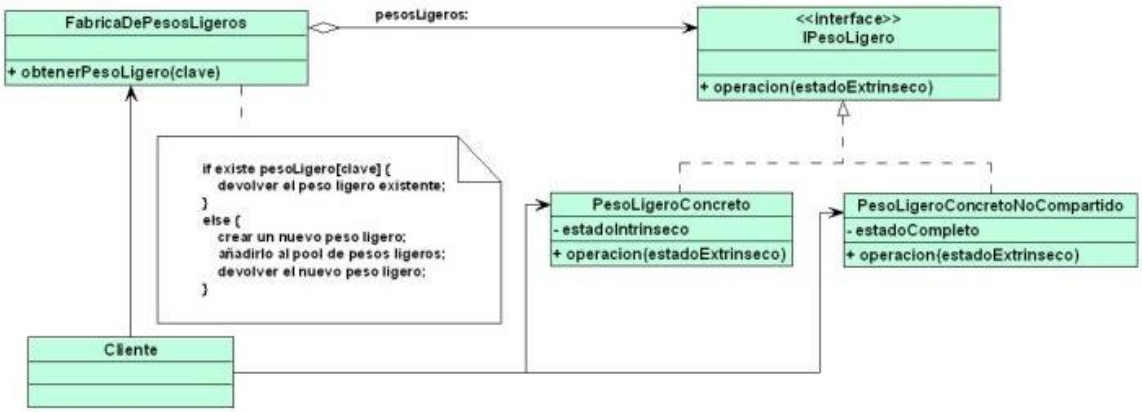
Una gran cantidad de objetos comparte un mismo objeto con propiedades comunes con el fin de ahorrar memoria.

USO.

Este patrón resulta tremendamente útil para evitar crear un gran número de objetos similares, mejorando con ello el rendimiento de la aplicación.

No se trata de crear muchos objetos de forma dinámica, sino de crear sólo un objeto intermedio para cada entidad concreta.

DISEÑO.



CODIGO.

```java
package estructurales.flyweight.flyweight01;

public class Main
{
    public static void main(String[] args)
    {
        FabricaDeLineas fabrica = new FabricaDeLineas();

        ILineaLigera linea1 = fabrica.getLine( "AZUL" );
        ILineaLigera linea2 = fabrica.getLine( "ROJO" );
        ILineaLigera linea3 = fabrica.getLine( "AMARILLO" );
        ILineaLigera linea4 = fabrica.getLine( "AZUL" );

         System.out.println("--------------------");

         //can use the lines independently
         linea1.dibujar( 100, 400 ) ;
         linea2.dibujar( 200, 500 );
         linea3.dibujar( 300, 600 );
         linea4.dibujar( 400, 700 );
    }
}
```

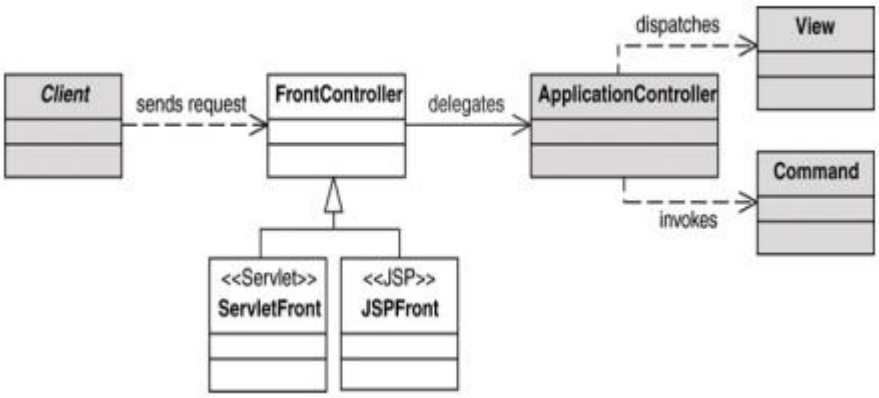
❖FRONT CONTROLER

La gestión de las peticiones HTTP a nuestra aplicación web puede ser centralizada o distribuida. Tener distribuida la gestión de peticiones en nuestras páginas JSP lleva a aplicaciones de baja calidad, en las que se genera mucho código similar distribuido de forma repetida por todas nuestras páginas (vistas)

USO.

utilizar un controlador como punto inicial para la gestión de las peticiones. El controlador gestiona se encarga de gestionar estas peticiones, y realizar algunas funciones como: comprobación de restricciones de seguridad, manejo de errores, mapeos y delegación de las peticiones a otros componentes de la aplicación que se encargarán de generar la vista adecuada para el usuario.

DISEÑO.


dispatches
View
Client
sends request
FrontController
delegates
ApplicationController
Command
invokes
<<Servlet>>
ServletFront
<<JSP>>
JSPFront

CODIGO.

```
public class EmployeeController extends HttpServlet {
// Initializes the servlet.
 public void init(ServletConfig config) throws
ServletException {
super.init(config);
 }
// Destroys the servlet.
 public void destroy() {
 }
 /** Processes requests for both HTTP
 * <code>GET</code> and <code>POST</code> methods.
 * @param request servlet request
 * @param response servlet response
 */
 protected void processRequest(HttpServletRequest
request, HttpServletResponse response)
throws ServletException, java.io.IOException {
String page;
/**ApplicationResources provides a simple API
 * for retrieving constants and other
 * preconfigured values**/
ApplicationResources resource =
 ApplicationResources.getInstance();
try {
// Use a helper object to gather parameter
// specific information.
 RequestHelper helper = new RequestHelper(request);
 Command cmdHelper= helper.getCommand();
// Command helper perform custom operation
page = cmdHelper.execute(request, response);
}
```

```
catch (Exception e) {
 LogManager.logMessage(
"EmployeeController:exception : " +
e.getMessage());
 request.setAttribute(resource.getMessageAttr(),
"Exception occurred : " + e.getMessage());
 page = resource.getErrorPage(e);
}
// dispatch control to view dispatch(request, response, page);
 }
 /** Handles the HTTP <code>GET</code> method.
 * @param request servlet request
 * @param response servlet response
 */
 protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, java.io.IOException {
 processRequest(request, response);
 }
 /** Handles the HTTP <code>POST</code> method.
 * @param request servlet request
 * @param response servlet response
 */
 protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, java.io.IOException {
processRequest(request, response);
 }
 /** Returns a short description of the servlet */
 public String getServletInfo() {
return "Front Controller Pattern" +
 " Servlet Front Strategy Example";
 }
```

```
 protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, java.io.IOException {
 processRequest(request, response);
 }
 /** Handles the HTTP <code>POST</code> method.
 * @param request servlet request
 * @param response servlet response
 */
 protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, java.io.IOException {
processRequest(request, response);
 }
 /** Returns a short description of the servlet */
 public String getServletInfo() {
return "Front Controller Pattern" +
 " Servlet Front Strategy Example";
 }
 protected void dispatch(HttpServletRequest request,
HttpServletResponse response,
String page)
 throws javax.servlet.ServletException,
java.io.IOException {
RequestDispatcher dispatcher =
 getServletContext().getRequestDispatcher(page);
dispatcher.forward(request, response);
 }
}
```

❖ PROXY

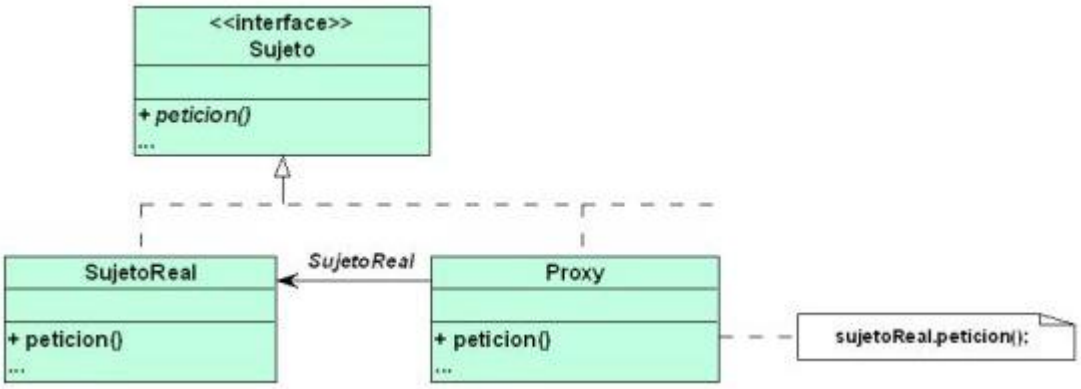
Este patrón se basa en proporcionar un objeto que haga de intermediario (proxy) de otro, para controlar el acceso a él.

USO.

- Proxy remoto: proporciona un representante local de un objeto situado en otro espacio de direcciones (en otro dispositivo conectado en red).
- Proxy virtual: usados para crear objetos costosos sólo cuando se soliciten.
- Proxy de protección: permiten controlar el acceso a un objeto cuando es accesible o no, dependiendo de determinados permisos.

- Referencia inteligente: un sustito de un puntero, que realiza operaciones adicionales en el momento de accederse al objeto.

DISEÑO.



CODIGO.

```
01.  package estructurales.proxy.proxy01;
02.
03.  public class Main
04.  {
05.      public static void main( String[] args )
06.      {
07.          // Abrimos un documento que puede contener una sola imagen
08.          Documento doc = new Documento( "Presupuesto.doc");
09.
10.          try     // Necesario al usar Thread
11.          {
12.              // Pausamos la ejecución del programa durante 3 segundos
13.              Thread.sleep( 3000 );
14.
15.              // Simulamos que el usuario hace scroll
16.              doc.hacerScroll();
17.          }
18.          catch( Exception e )
19.          {
20.              System.out.println("Ha ocurrido un error");
21.          }
22.      }
23.  }
```

Comportamiento.

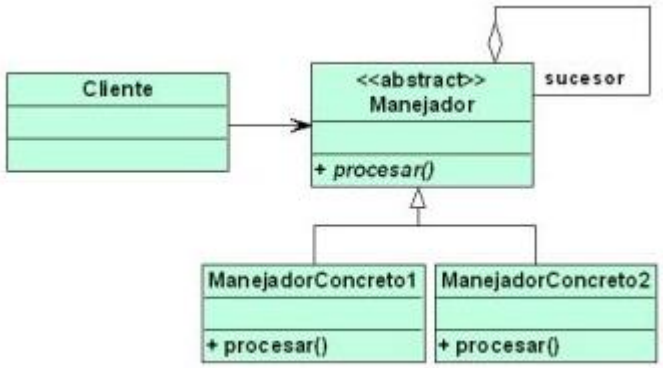
❖CHAIN OF RESPONSABILITY

SE EVITA ACOPLAR AL EMISOR Y RECEPTOR DE UNA PETICIÓN DANDO LA POSIBILIDAD A VARIOS RECEPTORES DE CONSUMIRLO. CADA RECEPTOR TIENE LA OPCIÓN DE CONSUMIR ESA PETICIÓN O PASÁRSELO AL SIGUIENTE DENTRO DE LA CADENA.

USO.

ESTE PATRÓN PUEDE RESULTARNOS ÚTIL EN CASOS EN LOS QUE UN OBJETO EMISOR DE UNA PETICIÓN DESCONOZCA QUÉ OBJETO(S) PODRÁ(N) ATENDER A LA MISMA.

DISEÑO.



CODIGO.

```
01.  package ChainOfResponsability1;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          Manejador m1 = new ManejadorAprobado();
08.          Manejador m2 = new ManejadorDenegado();
09.          Manejador m3 = new ManejadorPendiente();
10.
11.          m1.setSiguiente( m2 );
12.          m2.setSiguiente( m3 );
13.
14.          m1.comprobar("APROBADO");
15.          m1.comprobar("APROBADO");
16.          m1.comprobar("DENEGADO");
17.          m1.comprobar(null);
18.          m1.comprobar("DENEGADO");
19.      }
20.  }
```

❖COMMAND

Son objetos que encapsulan una acción y los parámetros que necesitan para ejecutarse.

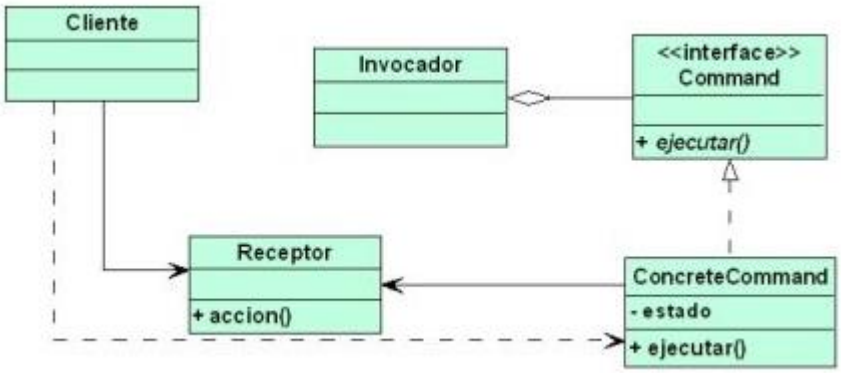
USO.

Este patrón resulta útil en escenarios en los que se han de enviar peticiones a otros objetos sin saber qué operación se ha de realizar, y ni tan siquiera quién es el receptor de dicha petición.

Dicho de otro modo: tenemos varios objetos que realizan acciones similares de forma diferente, y queremos que se procese la adecuada dependiendo del objeto solicitado.

El Invocador no sabe quién es el Receptor ni la acción que se realizará, tan sólo invoca un Comando que ejecuta la acción adecuada.

DISEÑO.



CODIGO.

```java
package Command;

public class Main
{
    public static void main(String[] args)
    {
        // Crear el objeto Menú (el Invocador)
        Menu objMenu = new Menu();

        // Crear el Receptor
        Receptor objReceptor = new Receptor();

        // Crear las opciones de menú, indicándoles el Receptor
        IMenuItem objOpcionAbrir = new MenuItemAbrir( objReceptor );
        IMenuItem objOpcionImprimir = new MenuItemImprimir( objReceptor );
        IMenuItem objOpcionSalir = new MenuItemSalir( objReceptor );

        // Agregar las opciones al Menú
        objMenu.add( objOpcionAbrir );
        objMenu.add( objOpcionImprimir );
        objMenu.add( objOpcionSalir );

        // Ejecutar cada opción del menú
        objMenu.get(0).ejecutar();
        objMenu.get(1).ejecutar();
        objMenu.get(2).ejecutar();
    }
}
```

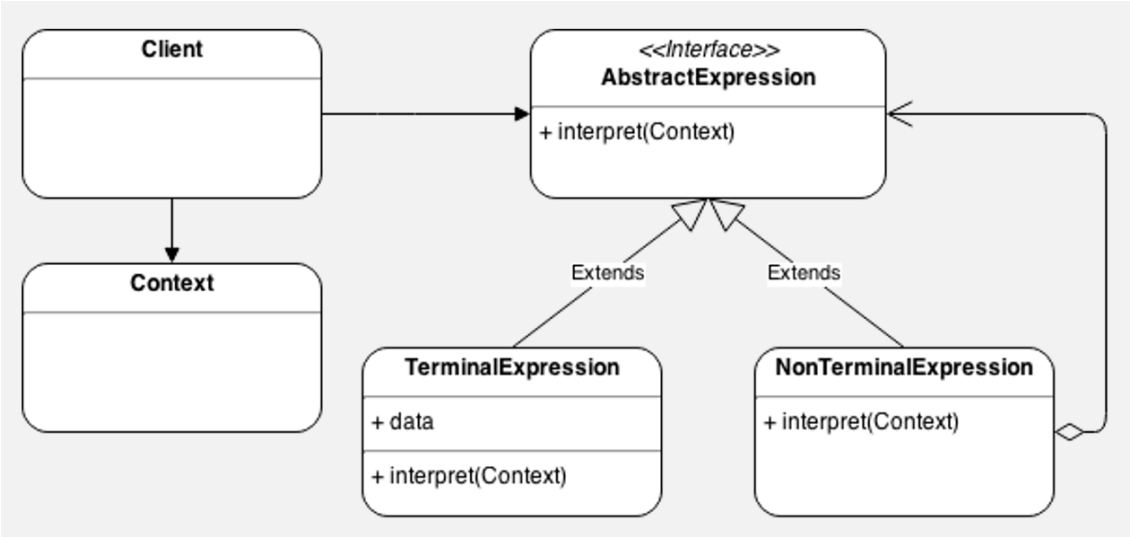
❖INTERPRETER

Define una representación para una gramática, así como el mecanismo para evaluarla. El árbol de sintaxis del lenguaje se suele modelar mediante el patrón Composite.

USO.

Es utilizado para evaluar un lenguaje definido como Expresiones, este patrón nos permite interpretar un lenguaje como Java, C#, SQL o incluso un lenguaje inventado por nosotros el cual tiene un significado; y darnos una respuesta tras evaluar dicho lenguaje.

DISEÑO.



CODIGO.

```
public class Interpreter{

        public static void main(String [] args)

        {

                // Creamos el arbol de expresiones y el contexto

                ArrayList tree = new ArrayList();

                Context context = new Context();


                // Añadimos los tokens pasados como argumentos

                for(String token : args)

                {

                        if(context.getInteger(token) >= 0)

                                tree.add(new NumericExpression(token));

                        else

                                tree.add(new OperationExpression(token));

                }
```

```
                // Interpretamos cada expresión

                for(Expression e : tree)

                        e.interpret(context);


                // Mostramos el resultado

                System.out.println("El resultado de la interpretación es " + context.getResult());

        }

}
```

❖ITERATOR

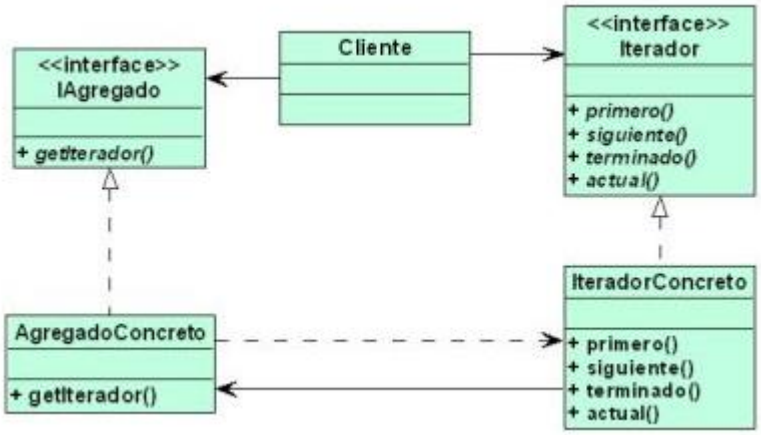
Se utiliza para poder movernos por los elementos de un conjunto de forma secuencial sin necesidad de exponer su implementación específica.

USO.

Según el libro de GoF, podemos utilizarlo cuando necesitemos recorrer secuencialmente los objetos de un elemento agregado sin exponer su representación interna.

Así pues, este patrón de diseño nos resultará útil para acceder a los elementos de un array o colección de objetos contenida en otro objeto

DISEÑO.



CODIGO.

```java
package Iterator;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            // Crear el objeto agregado que contiene la lista (un vector en este ejemplo)
            AgregadoConcreto agregado = new AgregadoConcreto();

            // Crear el objeto iterador para recorrer la lista
            Iterador iterador = agregado.getIterador();

            // Mover una posición adelante y mostrar el elemento
            String obj = (String) iterador.primero();
            System.out.println( obj );

            // Mover dos posiciones adelante
            iterador.siguiente();
            iterador.siguiente();

            // Mostrar el elemento actual
            System.out.println( (String) iterador.actual() + "\n" );

            // Volver al principio
            iterador.primero();

            // Recorrer todo
            while( iterador.hayMas() == true ) {
                System.out.println( iterador.siguiente() );
            }
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }
}
```

❖ MEDIATOR

Objeto que encapsula cómo otro conjunto de objetos interactúa y se comunican entre sí.

USO.

Una de las ventajas que ofrece la programación orientada a objetos (POO) es la posibilidad de reutilizar el código fuente, pero a medida que creamos objetos que se interrelacionan entre sí es menos probable que un objeto pueda funcionar sin la ayuda de otros.

Para evitar esto podemos utilizar el patrón Mediator, en el que se define una clase que hará de mediadora encapsulando la comunicación entre los objetos, evitándose con ello la necesidad de que lo hagan directamente entre sí.

DISEÑO.

CODIGO.

```
01.  package Mediator1;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          // Crear el objeto centralizador de la comunicación
08.          Mediador m = new Mediador();
09.
10.          // Crear los objetos que participarán en la comunicación
11.          Colega cc1 = new ColegaConcreto1( m );
12.          Colega cc2 = new ColegaConcreto2( m );
13.          Colega cc3 = new ColegaConcreto3( m );
14.
15.          // Agregarlos al objeto centralizador
16.          m.agregarColega( cc1 );
17.          m.agregarColega( cc2 );
18.          m.agregarColega( cc3 );
19.
20.          // Provocar un cambio en un uno de los elementos
21.          cc2.comunicar("ColegaConcreto2 ha cambiado!");
22.      }
23.  }
```

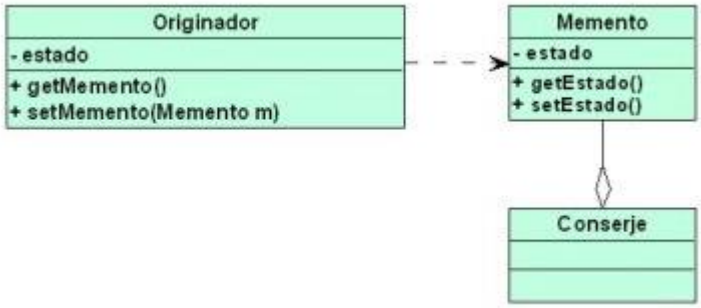
❖MEMENTO

Este patrón otorga la capacidad de restaurar un objeto a un estado anterior

USO.

Este patrón de diseño es útil cuando manejamos un objeto que necesitaremos restaurar a estados anteriores (como por ejemplo cuando utilizamos la función de deshacer en un procesador de textos) .

DISEÑO.



CODIGO.

```
01.  package Memento;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          // Crear el objeto originador/creador
08.          Originator creador = new Originator("Pedro", "Gil Mena");
09.
10.          // Crear el objeto gestor/vigilante del Memento
11.          Caretaker vigilante= new Caretaker();
12.
13.          // Crear el Memento y asociarlo al objeto gestor
14.          vigilante.setMemento( creador.createMemento() );
15.
16.          // Mostrar los datos del objeto
17.          System.out.println("Nombre completo: [" + creador.getNombre() + " " + creador.getApellidos() + "]" );
18.
19.          // Modificar los datos del objeto
20.          creador.setNombre("María");
21.          creador.setApellidos("Mora Miró");
22.
23.          // Mostrar los datos del objeto
24.          System.out.println("Nombre completo: [" + creador.getNombre() + " " + creador.getApellidos() + "]" );
25.
26.          // Restaurar los datos del objeto
27.          creador.setMemento( vigilante.getMemento() );
28.
29.          // Mostrar los datos del objeto
30.          System.out.println("Nombre completo: [" + creador.getNombre() + " " + creador.getApellidos() + "]" );
31.      }
32.  }
```

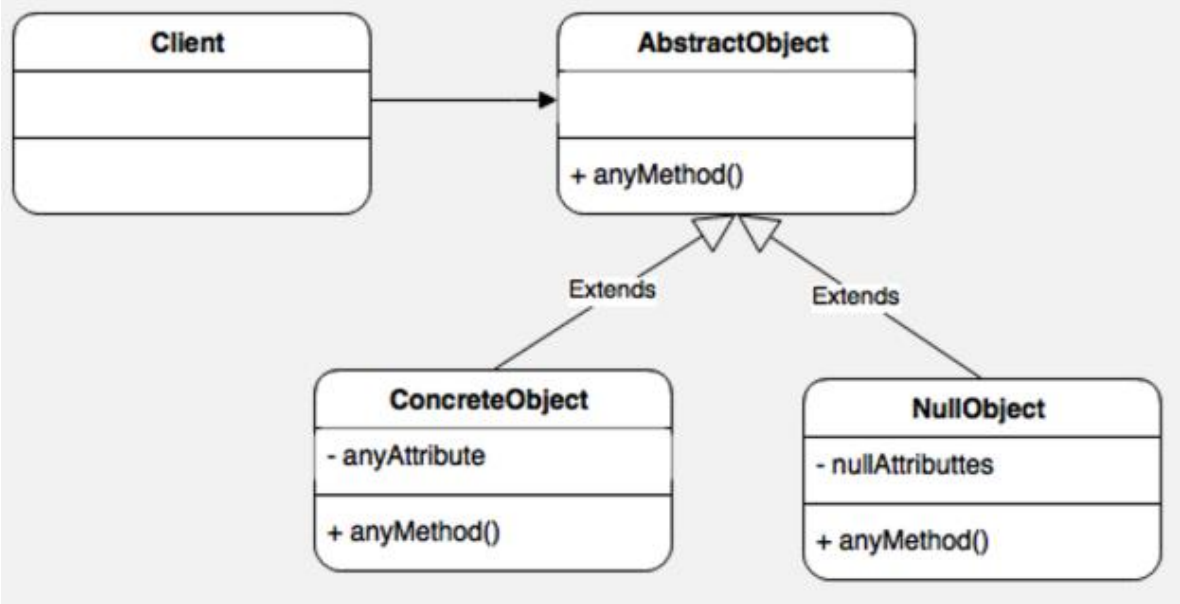
❖NULL OBJECT

El patrón de diseño Null Object nace de la necesidad de evitar los valores nulos que puedan originar error en tiempo de ejecución.

USO.

propone es utilizar instancias que implementen la interface requerida pero con un cuerpo vacío en lugar de regresar un valor null.

DISEÑO.



CODIGO.

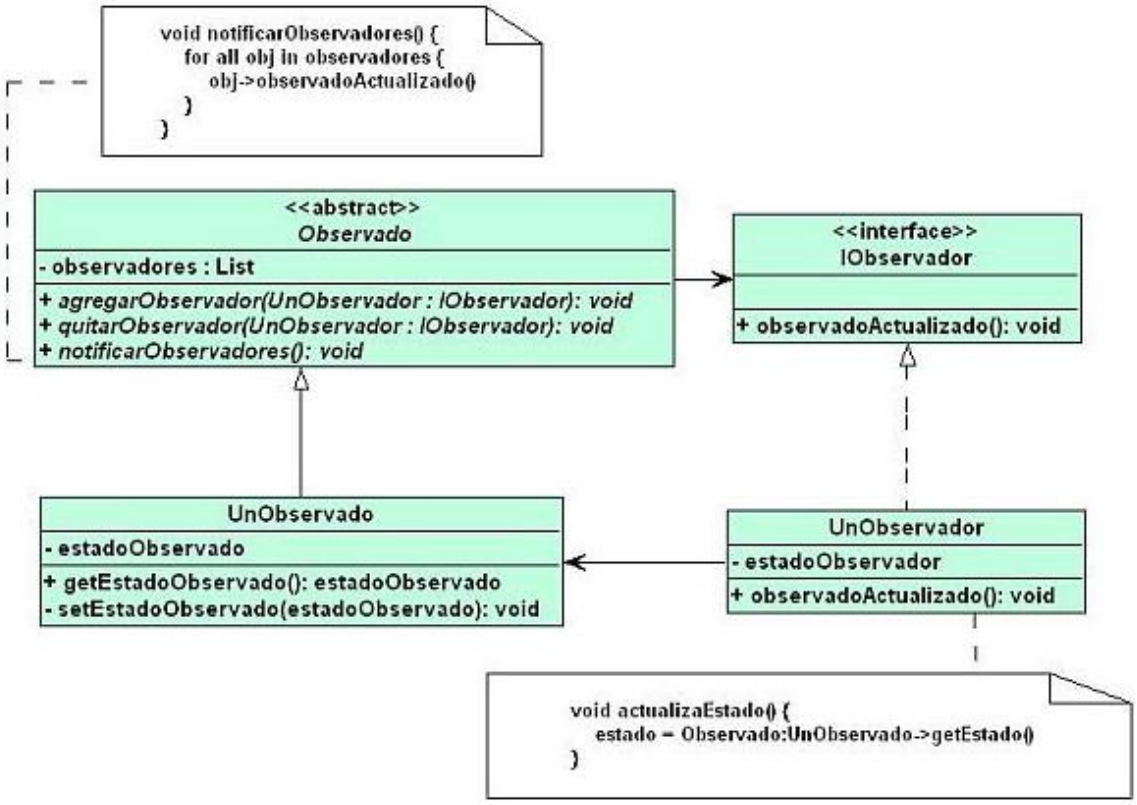
❖OBSERVER

Los objetos son capaces de suscribirse a una serie de eventos que otro objetivo va a emitir, y serán avisados cuando esto ocurra.

USO.

El patrón Observer puede ser utilizado cuando hay objetos que dependen de otro, necesitando ser notificados en caso de que se produzca algún cambio en él.

DISEÑO.



Se utilizarán los métodos agregarObservador( ) y quitarObservador( ) de la clase abstracta UnObservado para registrar en una lista qué objetos de tipo UnObservador deberán ser notificados o dejar de serlo cuando se produzca algún cambio en él (en tal caso recorrerá dicha lista para enviar una notificación a cada uno de ellos) .

El mensaje será enviado a UnObservador (que implementa la interface IObservador) utilizando su método observadoActualizado( ) .

Veamos a continuación un ejemplo sencillo en lenguaje Java en el cual cuando se produce un cambio en el observado éste envía una notificación a los observadores, que simplemente mostrarán un mensaje al recibirla.

CODIGO.

```
01.  package Observer;
02.
03.  public class Main
04.  {
05.      public static void main(String[] args)
06.      {
07.          // Instanciar el objeto que será Observado
08.          UnObservado objObservado = new UnObservado();
09.
10.          // Instanciar y registrar un Observador
11.          UnObservador objObservadorPepe  = new UnObservador("Pepe");
12.          objObservado.agregarObservador(objObservadorPepe);
13.
14.          // Instanciar y registrar otro Observador
15.          UnObservador objObservadorJuan  = new UnObservador("Juan");
16.          objObservado.agregarObservador(objObservadorJuan);
17.
18.          // Instanciar y registrar otro Observador
19.          UnObservador objObservadorMarta = new UnObservador("Marta");
20.          objObservado.agregarObservador(objObservadorMarta);
21.      }
22.  }
```

Concurrencia.

❖ACTIVE OBJECT

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.

USO.

- 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.

CODIGO.

```
clase  MyActiveObject  {

    privado  doble  val  =  0.0 ;
    private  BlockingQueue < Runnable >  dispatchQueue  =  new  LinkedBlockingQueue < Runnable > ();

    public  MyActiveObject ()  {
        new  Thread  ( new  Runnable ()  {

                @Override
                public  void  run ()  {
                    while  ( true )  {
                        try  {
                            dispatchQueue . tomar (). ejecutar ();
                        }  catch  ( InterruptedException  e )  {
                            // está bien, solo termina el despachador
                        }
                    }
                }
            }
        ). inicio ();
    }

    void  doSomething ()  lanza  InterruptedException  {
        dispatchQueue . put ( new  Runnable ()  {
                @Override
                public  void  run ()  {
                    val  =  1.0 ;
                }
            }
        );
    }
```

```
        void  doSomethingElse ()  lanza  InterruptedException  {
            dispatchQueue . put ( new  Runnable ()  {
                    @Override
                    public  void  run ()  {
                        val  =  2.0 ;
                    }
                }
            );
        }
    }
```

❖BALKING

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

USO.

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

CODIGO.

```java
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;
        }
    }
}
```

❖BINDIND PROPERTIES

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.

USO.

- Hay dos tipos de encuadernación. El enlace unidireccional debe aplicarse cuando una de las propiedades es de solo lectura. En otros casos, se debe aplicar la unión bidireccional.
- Se pueden eliminar bucles infinitos bloqueando la señal, o comparando el valor asignado con el valor de la propiedad antes de la asignación, o eliminando asignaciones innecesarias.

CODIGO.

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);
  }
}
```

[2]

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);
}
```

[3]

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);
}
```

❖COMPUTE KERNEL

ES UNA RUTINA COMPILADA PARA ACELERADORES DE ALTO RENDIMIENTO (COMO UNIDADES DE PROCESAMIENTO DE GRÁFICOS (GPU), PROCESADORES DE SEÑAL DIGITAL (DSP) O MATRICES DE COMPUERTAS PROGRAMABLES EN CAMPO (FPGA)), SEPARADAS DE PERO UTILIZADAS POR UN PROGRAMA PRINCIPAL (NORMALMENTE SE EJECUTA EN UNA UNIDAD CENTRAL DE PROCESAMIENTO). A VECES SE DENOMINAN SOMBREADORES DE CÓMPUTO, QUE COMPARTEN UNIDADES DE EJECUCIÓN CON SOMBREADORES DE VÉRTICES Y SOMBREADORES DE PÍXELES EN GPU, PERO NO SE LIMITAN A LA EJECUCIÓN EN UNA CLASE DE DISPOSITIVO O API DE GRÁFICOS.

USO.

- LOS NÚCLEOS DE CÓMPUTO CORRESPONDEN APROXIMADAMENTE A LOS BUCLES INTERNOS CUANDO SE IMPLEMENTAN ALGORITMOS EN LENGUAJES TRADICIONALES (EXCEPTO QUE NO HAY

una operación secuencial implícita), o al código que se pasa a los iteradores internos.

- Pueden especificarse mediante un lenguaje de programación separado como "OpenCL C" (administrado por la API de OpenCL), como "sombreadores de cómputo" (administrados por una API de gráficos como OpenGL), o incrustados directamente en el código de la aplicación escrito en un alto nivel lenguaje, como en el caso de C++ AMP.

❖MONITOR

es una clase, un objeto o un módulo seguro para subprocesos que se envuelve alrededor de un mutex para permitir el acceso seguro a un método o variable por más de un subproceso. La característica definitoria de un monitor es que sus métodos se ejecutan con exclusión mutua: en cada momento, a lo sumo, un hilo puede estar ejecutando cualquiera de sus métodos.

USO.

Un monitor consta de un objeto mutex (bloqueo) y variables de condición. Una variable de condición esencialmente es un contenedor de subprocesos que esperan una determinada condición. Los monitores proporcionan un mecanismo para que los subprocesos otorguen temporalmente acceso exclusivo para esperar a que se cumpla alguna condición, antes de recuperar el acceso exclusivo y reanudar su tarea.

CODIGO.

```
acquire(m); // Acquire this monitor's lock.
while (!p) { // While the condition/predicate/assertion that we are waiting for is not true...
    wait(m, cv); // Wait on this monitor's lock and condition variable.
}
// ... Critical section of code goes here ...
signal(cv2); -- OR -- notifyAll(cv2); // cv2 might be the same as cv or different.
release(m); // Release this monitor's lock.
```

```
// ... (previous code)
// About to enter the monitor.
// Acquire the advisory mutex (lock) associated with the concurrent
// data that is shared between threads,
// to ensure that no two threads can be preemptively interleaved or
// run simultaneously on different cores while executing in critical
// sections that read or write this same concurrent data. If another
// thread is holding this mutex, then this thread will be put to sleep
// (blocked) and placed on m's sleep queue.  (Mutex "m" shall not be
// a spin-lock.)
acquire(m);
// Now, we are holding the lock and can check the condition for the
// first time.

// The first time we execute the while loop condition after the above
// "acquire", we are asking, "Does the condition/predicate/assertion
// we are waiting for happen to already be true?"

while (!p())    // "p" is any expression (e.g. variable or
        // function-call) that checks the condition and
        // evaluates to boolean.  This itself is a critical
        // section, so you *MUST* be holding the lock when
        // executing this "while" loop condition!

// If this is not the first time the "while" condition is being checked,
// then we are asking the question, "Now that another thread using this
// monitor has notified me and woken me up and I have been context-switched
// back to, did the condition/predicate/assertion we are waiting on stay
// true between the time that I was woken up and the time that I re-acquired
// the lock inside the "wait" call in the last iteration of this loop, or
// did some other thread cause the condition to become false again in the
// meantime thus making this a spurious wakeup?
```

```
{
    // If this is the first iteration of the loop, then the answer is
    // "no" -- the condition is not ready yet. Otherwise, the answer is:
    // the latter.  This was a spurious wakeup, some other thread occurred
    // first and caused the condition to become false again, and we must
    // wait again.

    wait(m, cv);
        // Temporarily prevent any other thread on any core from doing
        // operations on m or cv.
        // release(m)       // Atomically release lock "m" so other
        //          // code using this concurrent data
        //          // can operate, move this thread to cv's
        //          // wait-queue so that it will be notified
        //          // sometime when the condition becomes
        //          // true, and sleep this thread. Re-enable
        //          // other threads and cores to do
        //          // operations on m and cv.
        //
        // Context switch occurs on this core.
        //
        // At some future time, the condition we are waiting for becomes
        // true, and another thread using this monitor (m, cv) does either
        // a signal/notify that happens to wake this thread up, or a
        // notifyAll that wakes us up, meaning that we have been taken out
        // of cv's wait-queue.
        //
        // During this time, other threads may cause the condition to
        // become false again, or the condition may toggle one or more
        // times, or it may happen to stay true.
        //
        // This thread is switched back to on some core.
```

```
        // acquire(m)       // Lock "m" is re-acquired.

    // End this loop iteration and re-check the "while" loop condition to make
    // sure the predicate is still true.

}

// The condition we are waiting for is true!
// We are still holding the lock, either from before entering the monitor or from
// the last execution of "wait".

// Critical section of code goes here, which has a precondition that our predicate
// must be true.
// This code might make cv's condition false, and/or make other condition variables'
// predicates true.

// Call signal/notify or notifyAll, depending on which condition variables'
// predicates (who share mutex m) have been made true or may have been made true,
// and the monitor semantic type being used.

for (cv_x in cvs_to_notify) {
    notify(cv_x); -- OR -- notifyAll(cv_x);
}
// One or more threads have been woken up but will block as soon as they try
// to acquire m.

// Release the mutex so that notified thread(s) and others can enter their critical
// sections.
release(m);
```

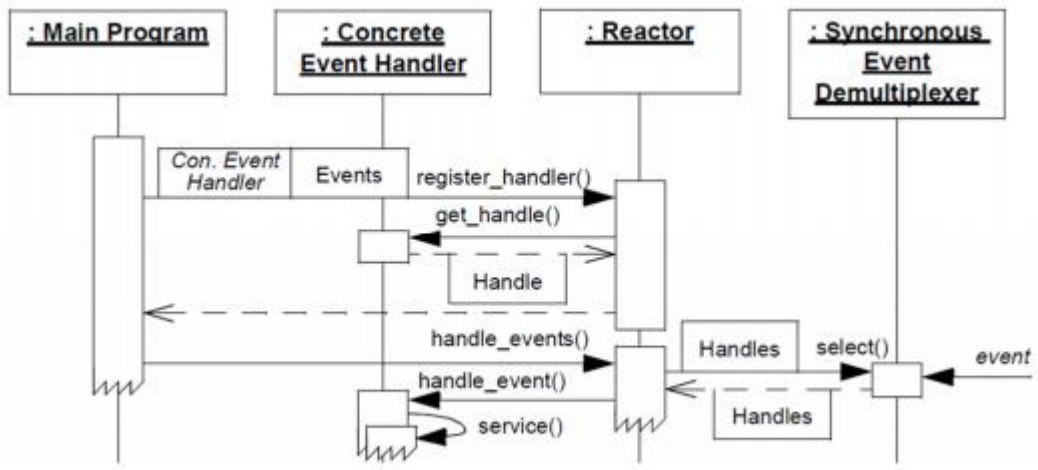
❖REACTOR

ES UN PATRÓN DE MANEJO DE EVENTOS PARA MANEJAR SOLICITUDES DE SERVICIO ENTREGADAS SIMULTÁNEAMENTE A UN MANEJADOR DE SERVICIO POR UNA O MÁS ENTRADAS. EL MANEJADOR DEL SERVICIO LUEGO DEMULTIPLEXA LAS SOLICITUDES ENTRANTES Y LAS ENVÍA SINCRÓNICAMENTE A LOS MANEJADORES DE SOLICITUDES ASOCIADOS.

USO.

EL PATRÓN DEL REACTOR SEPARA COMPLETAMENTE EL CÓDIGO ESPECÍFICO DE LA APLICACIÓN DE LA IMPLEMENTACIÓN DEL REACTOR, LO QUE SIGNIFICA QUE LOS COMPONENTES DE LA APLICACIÓN SE PUEDEN DIVIDIR EN PARTES MODULARES Y REUTILIZABLES.

DISEÑO.



CODIGO.

```
#include "Reactor.h"
#include <poll.h>
/* Other include files omitted... */

/* Bind an event handler to the struct used to interface poll(). */
typedef struct
{
   EventHandler handler;
   struct pollfd fd;
} HandlerRegistration;

static HandlerRegistration registeredHandlers[MAX_NO_OF_HANDLES];

/* Add a copy of the given handler to the first free position in registeredHandlers. */
static void addToRegistry(EventHandler* handler);

/* Identify the event handler in the registeredHandlers and remove it. */
static void removeFromRegistry(EventHandler* handler);

/* Implementation of the Reactor interface used for registrations.*/

void Register(EventHandler* handler)
{
   assert(NULL != handler);
   addToRegistry(handler);
}

void Unregister(EventHandler* handler)
{
   assert(NULL != handler);
   removeFromRegistry(handler);
}
```

❖LOCK

ES UN MECANISMO DE SINCRONIZACIÓN PARA IMPONER LÍMITES AL ACCESO A UN RECURSO EN UN ENTORNO DONDE HAY MUCHOS HILOS DE EJECUCIÓN.

USO.

Un bloqueo está diseñado para hacer cumplir una política de control de concurrencia de exclusión mutua.

CODIGO.

```
if  ( lock  ==  0 )  {
    // lock free,
    configúrelo lock  =  myPID ;
}
```

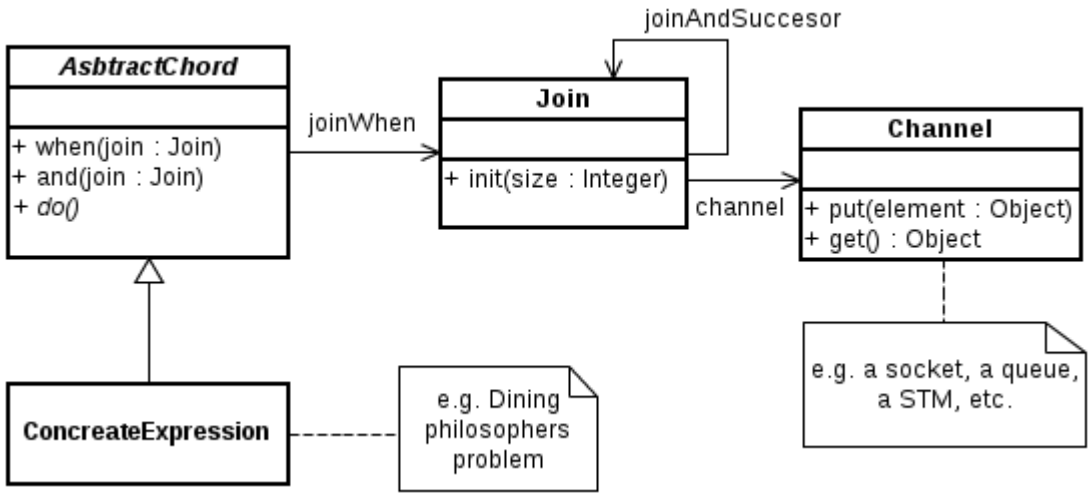
```
Cuenta de clase  pública // Este es un monitor de una cuenta { private decimal _balance = 0 ; objeto privado _balanceLock = nuevo objeto ();


    Depósito público nulo  ( cantidad decimal ) { // Solo un hilo a la vez puede ejecutar esta declaración. lock ( _balanceLock ) { _balance
+ = cantidad ; } }



   public  void  Withdraw ( cantidad decimal  ) { // Solo un hilo a la vez puede ejecutar esta declaración. lock ( _balanceLock ) { _balance
- = cantidad ; } } }
```

❖JOIN

Proporciona una forma de escribir programas informáticos concurrentes, paralelos y distribuidos mediante el paso de mensajes.

USO.

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.



CODIGO.

```
clase  JoinExample  {
    int  fragment1 ()  &  fragment2 ( int  x )  {
        //
        Devolverá el valor de x al llamante del fragmento1 return  x ;
    }
}
```

❖DOUBLE-CHECKED LOCKING

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.

USO.

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.

CODIGO.

```
// Versión de subproceso único
clase  Foo  {
    private  Helper  helper ;
    pública  ayudante  getHelper ()  {
        si  ( ayudante  ==  nula )  {
            ayudante  =  nueva  ayudante ();
        }
        ayudante de retorno  ; }


    // otras funciones y miembros ...
}
```

```
//
Clase de  versión multiproceso correcta pero posiblemente costosa Foo  {
    private  Helper  helper ;
    pública  sincronizado  ayudante  getHelper ()  {
        si  ( ayudante  ==  nula )  {
            ayudante  =  nueva  ayudante ();
        }
        ayudante de retorno  ; }


    // otras funciones y miembros ...
}
```