Desarrollo utilizando patrones software

Patrón de diseño:

- Resuelve un problema
- Concepto comprobado
- Soluciones no-obvias
- No solo describen módulos, también mecanismos y estructuras más profundas.
- Minimizan la intervención humana.

Clasificación: **tipo**

- **Patrones arquitectónico**s: Si nos fijamos en la estructura de los problemas.
- **Patrones de datos**: Problemas relacionados con la organización de datos.
- **Patrones de diseño orientado a objetos**: Cómo se intercomunican y se ubican en una clasificación jerárquica los subsistemas y componentes.
    - Patrones Creacionales: Creación, composición y representación de objetos.
    - Patrones Estructurales: Organización e integración (combinar objetos, hacer herencia, etc).
    - Patrones Comportamentales: Responsabilidades de cada objeto y cómo se comunican.
- **Patrones de diseño de interfaces**: Para realizar interfaces en función del contexto y de los usuarios.
- **Patrones Web**

Otro criterio de clasificación: **ámbito** (¿El patrón se aplica a clases u objetos?).

- Patrones de **clase**: Relaciones entre clases y objetos. Son relaciones estáticas. Suelen ser herencia (se fija en tiempo de compilación).
- Patrones de **objetos**: Relaciones entre objetos. Son relaciones dinámicas. Tiempo de ejecución.

Un patrón creacional de clase utiliza herencia para componer las clases. Un patrón creacional de objetos describe formas de componer y ensamblar objetos.

Marcos de Trabajo (framework)

- Sirve para proporcionar una infraestructura, esqueleto de la implementación de un sistema. Es la mínima arquitectura reutilizable que proporciona una estructura y

comportamiento sin dar un dominio. Es un esqueleto con enchufes o ganchos para adaptarlo a un dominio de problemas específico.
- En la POO un marco de trabajo es una colección de clases cooperantes.
- Se aplican sin cambios, se añaden elementos gracias a los ganchos.

Patrones de diseño vs Marcos de trabajo

- Los patrones son más abstractos.
- Los patrones son elemento arquitectónicos más pequeños.
- Los patrones son más especializados.

Programación Orientada a Objetos

- La **corrección** depende de: encapsulación, granularidad (tamaño, qué forma el objeto), dependencia, flexibilidad, rendimiento, evolución, reusabilidad, etc. Pueden entrar en conflicto. La POO ayuda a conseguirlos.
- Diferentes **estrategias**, todas con sus ventajas e inconvenientes.
- Los patrones de diseño nos ayudan a encontrar las soluciones menos obvias y los objetos que pueden representarlas mejor. Muchos de esos objetos no se encuentran en el análisis, sino en el proceso de hacer un diseño más flexible y reutilizable.
- Hay patrones que describen cómo representar subsistemas completos como objetos o cómo descomponer objetos en otros más pequeños y cómo tratarlos.
- **Interfaz** de un objeto: Nombre, los objetos que acepta y el valor de retorno, de todos los métodos que forman ese objeto. Los patrones de diseño nos ayudan a definir interfaces correctas, y puede obligar a que se cumplan relaciones.
- Un objeto es de un tipo cuando este acepta todas las llamadas que se pueden hacer a los objetos de ese tipo. Un objeto puede ser de muchos tipos a la vez, y tener el mismo tipo que otro objeto muy distinto (solo compartiendo esa parte de su interfaz).
- Las interfaces de los objetos pueden contener otras interfaces, entonces la contenida es un subtipo de la contenedora.
- **Ligadura dinámica**: Conexión entre la llamada a un objeto, y una de sus posibles operaciones. La llamada no se compromete con ninguna posible implementación. Se pueden sustituir los objetos durante la ejecución a otros con la misma interfaz (**polimorfismo**).

Lenguaje de patrones de diseño

Colección de patrones interrelacionado con el objeto a mostrar con el objetivo de mostrar cómo colaboran esos patrones para resolver un problema dentro de un dominio de aplicaciones.
Proporciona reglas para organizar los patrones.

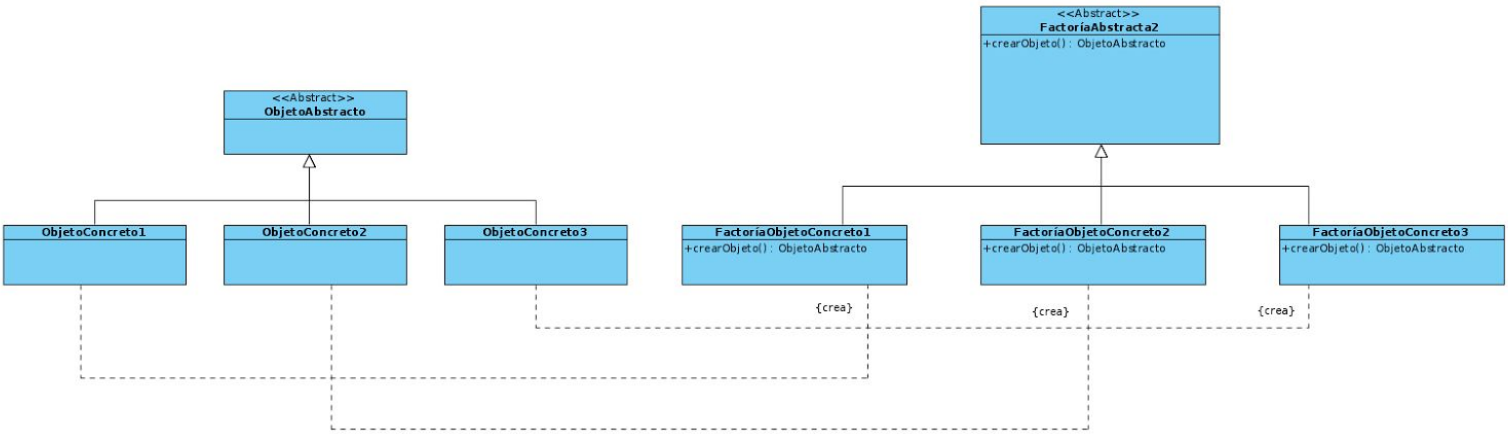
Patrones de diseño

Factoría abstracta (Abstract Factory)

Se trata de un patrón creacional cuyo objetivo es permitir crear objetos de un tipo no determinado, es decir, el marco de trabajo no tiene por qué saber a priori el tipo concreto de las instancias que va a crear. Es la aplicación quien decide de qué tipo se van a crear en cada momento.
La idea es poder añadir e instanciar nuevas clases específicas desde un marco de trabajo externo con los beneficios de un marco de trabajo no modificable.

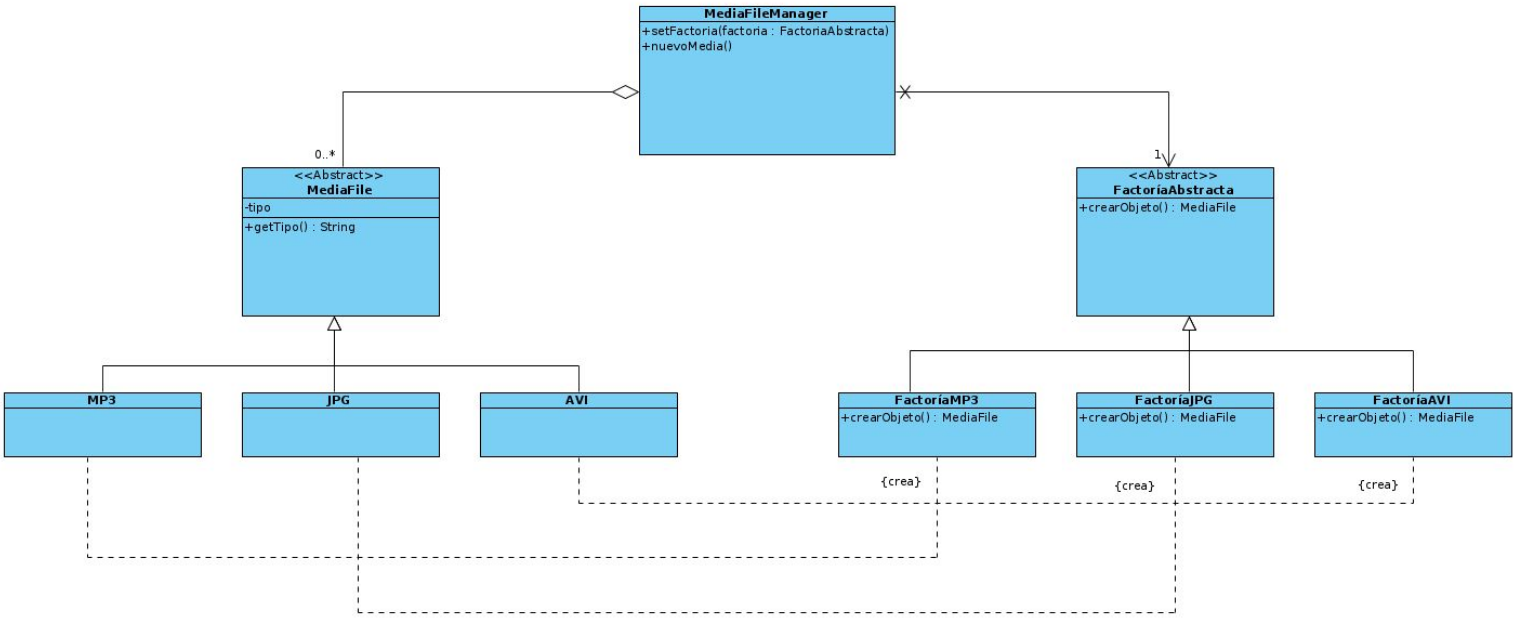
Cómo utilizarlo:

- Crear una interfaz denominada ObjetoAbstracto.
- Crear una interfaz FactoríaAbstracta.
    - Crear el método crearObjeto() el cual devuelve una instancia de ObjetoAbstracto.
- Crear el marco de trabajo (opcional, vamos a crearlo para entender cómo se usa):
    - Clase principal que trabajará con objetos de tipo ObjetoAbstracto.
    - Tiene acceso a una instancia de tipo FactoríaAbstrancta.
    - La clase principal tiene un método setFactoríaConcreta() para establecer la referencia del objeto de tipo FactoríaAbstracta.
- Crear una nueva clase que implementa a ObjetoAbstracto, llamada: ObjetoConcreto.
- Crear una nueva factoría que implemente a FactoríaAbstracta, esta vez de denominada FactoríaConcreta.
    - Implementar el método crearObjeto() de forma que cree y devuelva instancias de ObjetoConcreto.
- Podríamos crear tantas factorías concretas como objetos deseemos que se puedan crear.
- Durante la ejecución del programa, a través del método setFactoríaConcreta() hay que pasarle al marco de trabajo la factoría que será capaz de crear objetos de un tipo concreto (desconocido a priori) de ObjetoAbstracto.



Ejemplo:

- Clase MediaFile.
    - Tiene subclases que representan distintos formatos multimedia.
- Queremos crear una clase llamada MP3 y la clase MediaFileManager (marco de trabajo cerrado) debe poder instanciarla.
- Crear interfaz MediaFactoría (factoría abstracta de clases con distintos formatos multimedia).
    - Esta interfaz tiene el método crearObjeto() que devuelve una instancia de un MediaFile genérico.
- Crear clase factoría MP3Factoría.
    - Sirve para crear instancias de MP3 integradas dentro del marco de trabajo.
    - Implementa la interfaz MediaFactoria.
- Para crear archivos MP3 hay que llamar al método setMediaFactoría(...) de la clase MediaFileManager, el cual crea un enlace a una instancia de la clase factoría MP3.
- Cada vez que MediaFileManager necesita crear un objeto MediaFile, llama al método crearMedia() de MP3Factoría que devuelve un nuevo MP3.




```
// Objeto genérico (abstracto)
Class Abstract MediaFile{
        private String tipo;

        String getTipo(){
                return tipo;
        }

        private setTipo(String tipo){
                this.tipo = tipo;
        }
}
// Objetos concretos:
Class MP3 extend MediaFile{
        public MP3(){
                setTipo(“MP3”);
        }
}

Class JPG extend MediaFile{
        public MP3(){
                setTipo(“JPG”);
        }
}
```

```
Class AVI extend MediaFile{
	public MP3(){
		setTipo(“AVI”);
	}
}

// Factoría abstracta
Class Abstract FactoríaAbstracta{
	FileMedia crearObjeto();
}

// Factorías concretas
Class FactoríaMP3· extends FactoríaAbstracta{
	@Override
	FileMedia crearObjeto(){
		return new MP3;
	}
}

Class FactoríaJPG· extends FactoríaAbstracta{
	@Override
	FileMedia crearObjeto(){
		return new JPG;
	}
}

Class FactoríaAVI· extends FactoríaAbstracta{
	@Override
	FileMedia crearObjeto(){
		return new AVI;
	}
}

// Marco de trabajo que los utiliza
Class MediaFileManager{
	private FactoríaAbstracta factoría;
	private ArrayList<FileMedia> archivos;

	void setFactoria(FactoríaAbstracta factoría){
		this.factoria = factoria;
	}

	void nuevoMedia(){
		archivos.add(factoría.crearObjeto());
	}
```

```
}

//Main:
public static void main (String [ ] args) {
        FactoríaAbstracta factoríaMP3 = new FactoríaMP3();

        MediaFileManager.setFactoría(factoríaMP3);
        MediaFileManager.nuevoMedia();
        MediaFileManager.nuevoMedia();
        MediaFileManager.nuevoMedia();

        FactoríaAbstracta factoríaJPG = new FactoríaJPG();

        MediaFileManager.setFactoría(factoríaJPG);
        MediaFileManager.nuevoMedia();
}
```

Método factoría (Factory Method) + Diferencias con Abstract Factory

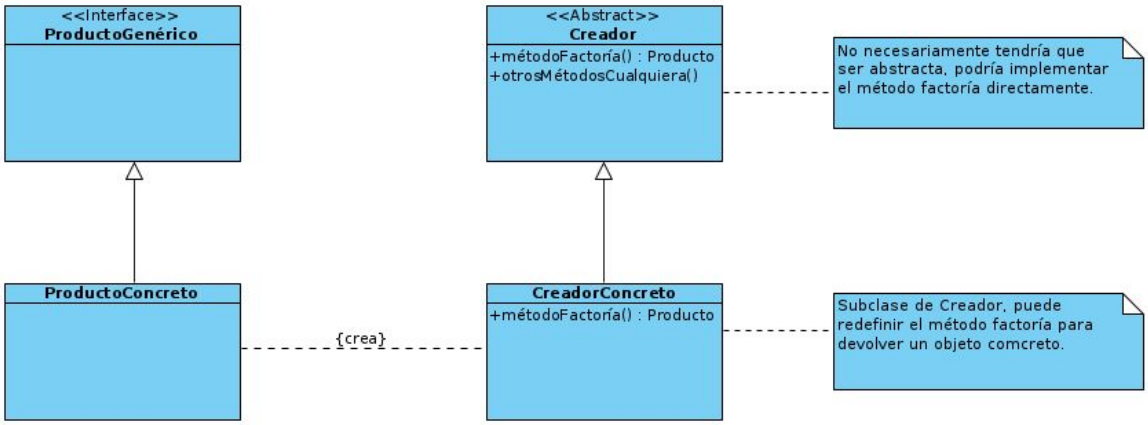
Se trata de un patrón creacional parecido a Factoría Abstracta.
La diferencia con la factoría abstracta es que esta vez se trata únicamente de un método, la factoría abstracta es un objeto. Al tratarse de un método puede ser redefinido en subclases. Factoría Abstracta sin embargo puede tener varios métodos factoría en él, cuando se usa este patrón se delega la responsabilidad de crear objetos en un método factoría del objeto factoría concreta. Por el contrario, en el patrón método factoría, se utiliza un método factoría para crearlo directamente sin tener que delegar al objeto factoría en concreto.
El objetivo de una clase de tipo FactoríaAbtracta es solo crear objetos, el objetivo de una clase con el método factoría puede ser muchos otros.

Cómo utilizarlo:

- Crear una clase con un método con capacidad de instanciar objetos (entre otros métodos). Ese es el método factoría.
    - Este método podría instanciar un tipo u otro en función de parámetros.
- Subclases de la clase con el método factoría pueden redefinirlo para devolver otro tipo concreto.

Ejemplo:

- Se desea obtener un objeto en función de una orden de un usuario.
- Un juego crea instancias de animales en función del país elegido por el usuario.
    - El método factoría simplemente podría estar parametrizado, para crear animales de un tipo u otro.
    - Si, además, hay alguna distinción, como por ejemplo una separación por continentes, podría hacerse una clase abstracta con el método factoría que sea redefinida por subclases concretas de cada continente, donde se siga teniendo un método factoría que responda en función del país (pero para ese continente concreto).

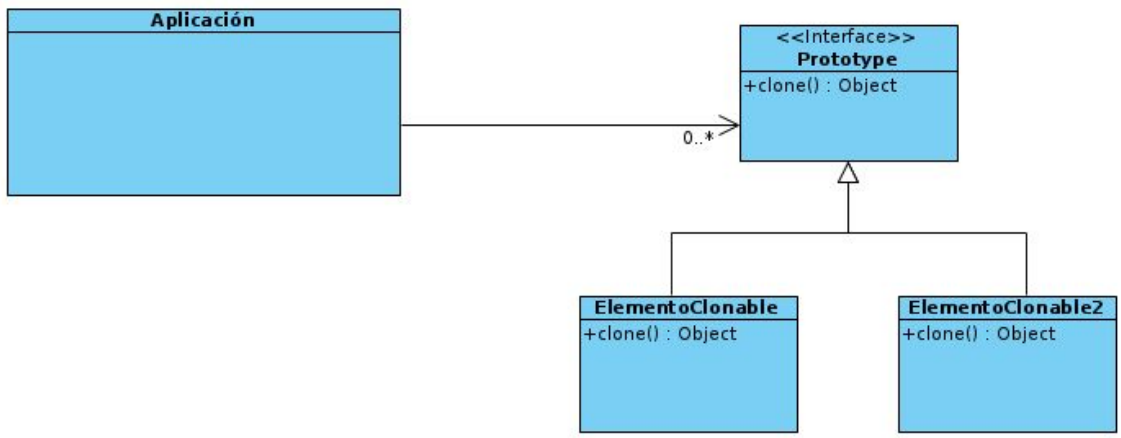

<<Interface>>
ProductoGenérico
<<Abstract>>
Creador
+métodoFactoría() : Producto
+otrosMétodosCualquiera()
No necesariamente tendría que ser abstracta, podría implementar el método factoría directamente.
ProductoConcreto
{crea}
CreadorConcreto
+métodoFactoría() : Producto
Subclase de Creador, puede redefinir el método factoría para devolver un objeto comcreto.

Prototipo (Prototype)

Se trata de un patrón creacional.
El coste de crear un nuevo objeto con el método new() puede ser inasumible para la aplicación, este método consiste en clonar instancias ya existentes para no tener que crearlas.

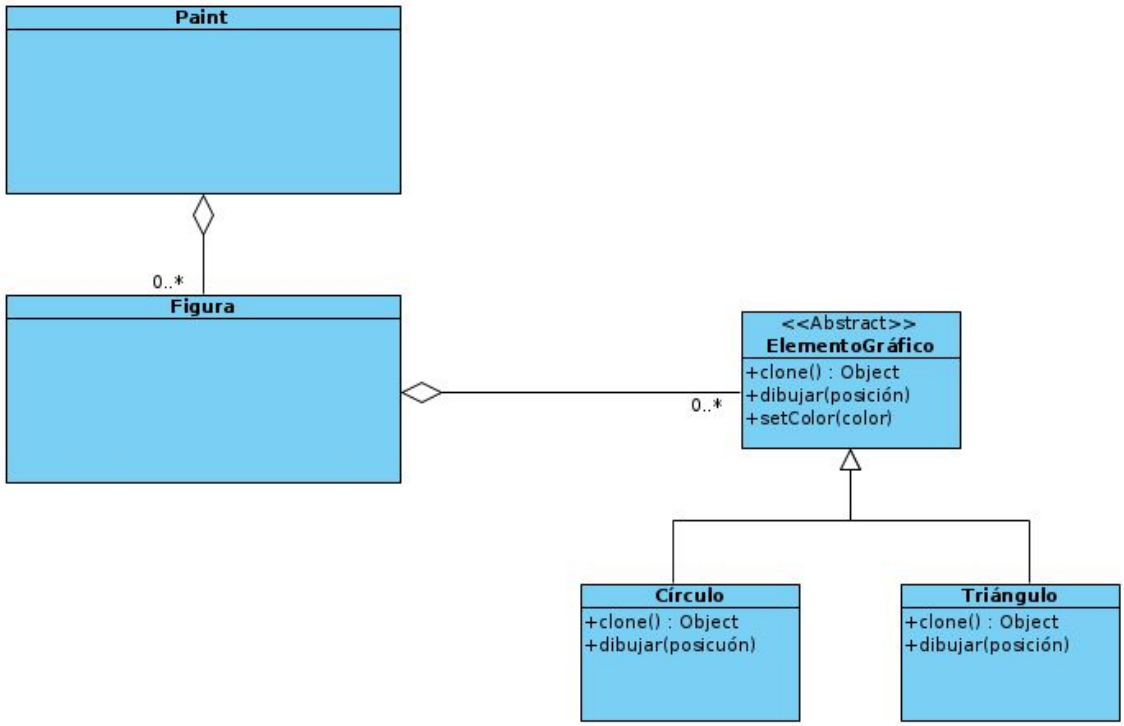
Cómo utilizarlo:

- Crear una interfaz Prototype con el método clone, el cual debe devolver un Object (en java esta interfaz tiene que implementar cloneable).
- Los objetos que heredan de Prototype han de implementar el método clone, permitiendo así su clonación devolviendo una nueva instancia igual a ellos.



Ejemplo:

- Una aplicación que maneja elementos gráficos, como un círculo o un triángulo. Estas figuras gráficas pueden clonarse para crear otras iguales.
- Los elementos gráficos implementan la clase abstracta ElementoGráfico, con el método clone. Esta clase abstracta ya contaba con otros métodos oportunos para los elementos gráficos.
- Cada elemento gráfico concreto define cómo se realiza su clonación, devolviendo el método clone una copia exacta de la instancia concreta.

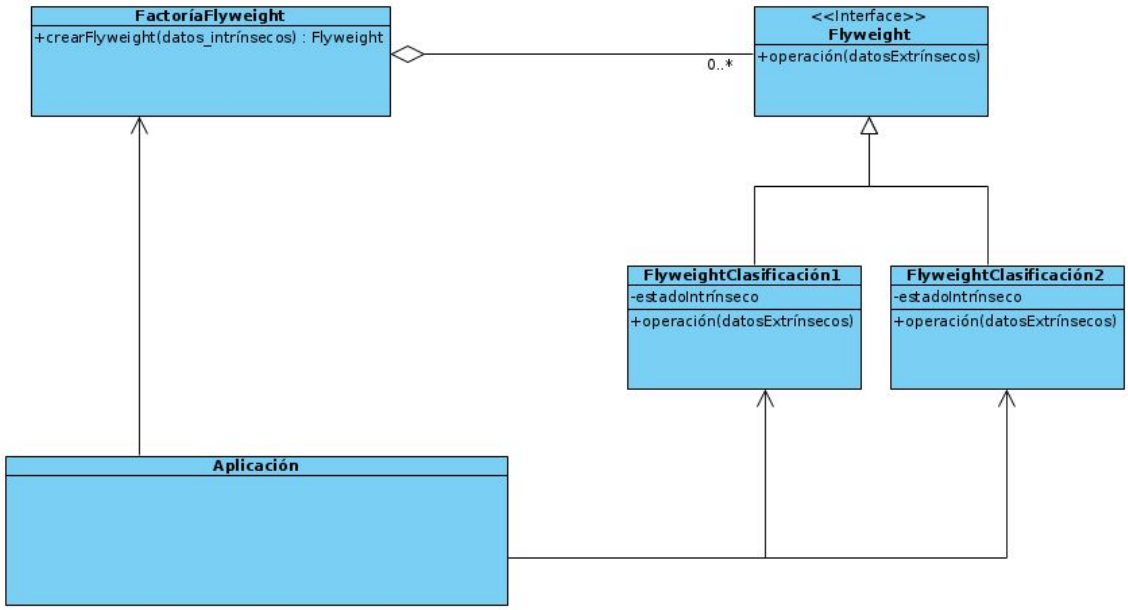

Paint
0..*
Figura
<<Abstract>>
ElementoGráfico
+clone() : Object
+dibujar(posición)
+setColor(color)
0..*
Círculo
+clone() : Object
+dibujar(posicuón)
Triángulo
+clone() : Object
+dibujar(posición)

Flyweight

El objetivo de este patrón es reducir el tamaño de memoria que necesita el programa para funcionar. Sirve para mejorar el rendimiento del software cuando tenemos un gran número de objetos ligeros.
La idea es almacenar los objetos ligeros (flywights) como uno solo. Para ello es necesario distinguir qué parte del estado de los objetos se comparte, y cuál no. Estos flywights tienen un estado extrínseco, el cual depende del contexto, que no es almacenado ya que es individual y se le indica cuando es necesario. Se almacena el estado intrínseco, el cual no depende del contexto y son compartidos por varios flyweights del mismo tipo.
Este patrón suele ir muy ligado a una factoría, la cual se encarga de instanciar flyweight únicamente cuando es necesario, indicándole sus datos extrínsecos. Los flyweight ya instanciados suelen almacenarse en la factoría, de este modo cuando se solicita uno de ellos, si ya está instanciado no es necesario volver a obtenerlo.
Puede haber objetos flyweight que no compartan el estado con los demás.

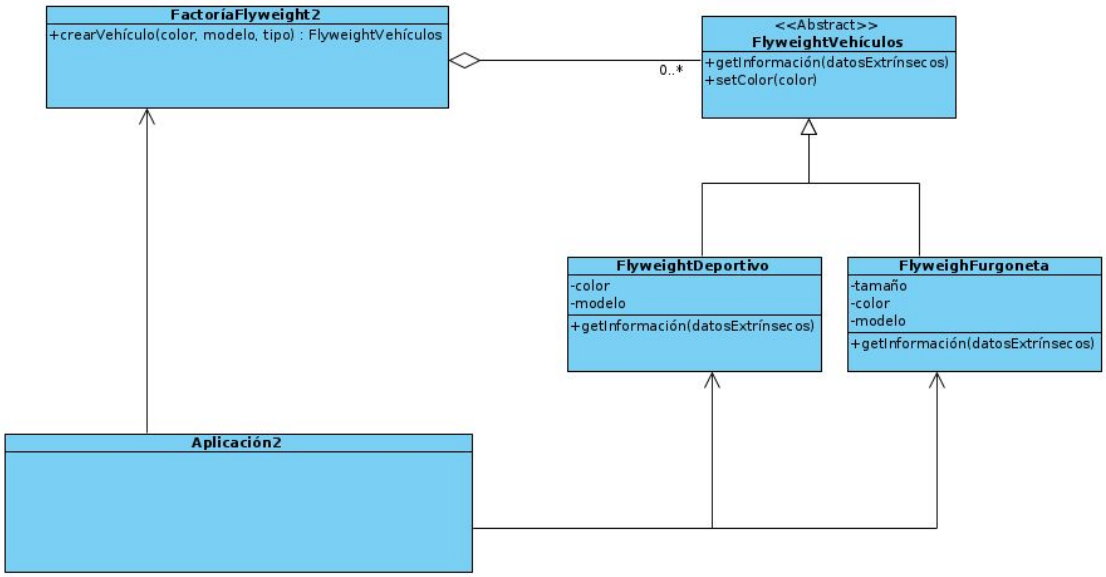
Cómo utilizarlo:

- Crear una interfaz denominada Flyweight, la cual puede contener varios métodos comunes, pero ha de contener al menos un método que dependa del contexto, es decir, parametrizado permitiendo indicar datos de un estado concreto. Ésta es la operación que nos permite indicar el estado extrínseco.
- Implementar Flyweight tantas veces como clasificaciones podamos crear con nuestros objetos ligeros. Por ejemplo, si se tratase de un sistema de gestión de vehículos en el que tenemos turismos y motocicletas, si todos los turismos pueden compartir unos datos comunes (estado intrínseco) que siempre es el mismo para todos ellos, pero no es el mismo para las motocicletas (que, además, tienen el suyo propio), tendríamos que crear dos clases que implementen Flyweight y que contengan los valores del estado intrínseco para cada tipo.
- Podemos implementar Flyweight especiales para objetos de esta clasificación que no compartan estado, pero sin romper la utilidad de este patrón. Siempre es mejor tener clasificaciones que engloben a un gran número de clases.
- Crear una clase factoría de flyweights la cual se encargue de crear, en función del estado extrínseco requerido los flyweights. Los flyweight instanciados se almacenan en una colección interna a la factoría, de modo que no es necesario volverlos a instanciar en caso de volverlos a necesitar. Esto último funciona con la idea de que, al ser tantos objetos ligeros, no siempre vamos a necesitar tenerlos todos cargados a la vez, y puede que a veces solo necesitemos algunos concretos, por lo que de este modo ahorramos memoria.
- Cuando se solicita a la factoría un flyweight con un estado INTRÍNSECO concreto, esta comprueba si ya existe y lo devuelve, en caso contrario lo crea y almacena.
- Desde la aplicación podemos crear objetos con los flyweight, los cuales están siendo compartidos de forma interna gracias a la factoría.



Ejemplos:

- Un sistema operativo de ventanas. Las ventanas tienen unos atributos comunes: color, tamaño, etc. En lugar de tener un objeto por ventana, podríamos tener una clase que almacene la información común y nos sirva para obtener la ventana deseada.
- Un sistema que almacena una serie de vehículos. Las furgonetas y los deportivos tienen atributos distintos los unos de los otros pero compartidos entre los del mismo tipo. Podríamos crear dos implementaciones de Flyweight que tengan como estado intrínseco datos respectivos al modelo de vehículo en función de si es una furgoneta o un deportivo, como la marca, color, etc. La factoría produciría objetos del tipo vehículo en la que todos los que compartan datos intrínsecos compartirían la misma instancia de flyweight, solo habría que indicar datos extrínsecos (como el número de matrícula) para crear cada instancia de tipo vehículo.


FactoríaFlyweight2
+crearVehículo(color, modelo, tipo) : FlyweightVehículos
0..*
<<Abstract>>
FlyweightVehículos
+getInformación(datosExtrínsecos)
+setColor(color)
FlyweightDeportivo
-color
-modelo
+getInformación(datosExtrínsecos)
FlyweighFurgoneta
-tamaño
-color
-modelo
+getInformación(datosExtrínsecos)
Aplicación2

Singleton

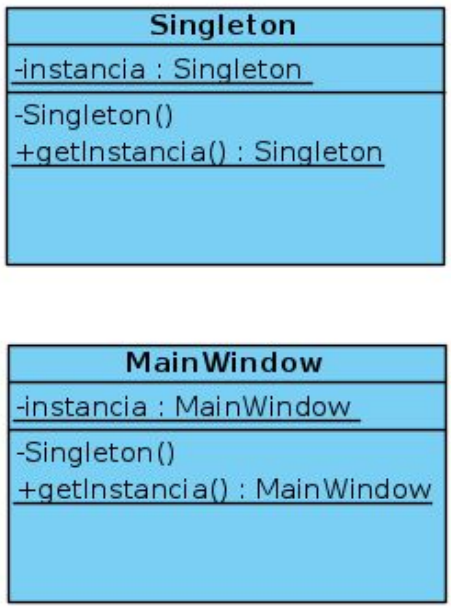
Se trata de un patrón creacional que restringe la instanciación a un único objeto. Nunca será posible crear más de una instancia, por lo que no puede haber un constructor público, pero a la vez es necesario que se pueda obtener la instancia desde las clases que se necesite.

Cómo utilizarlo:

- Crear una variable privada estática, llamada instancia, del tipo que define la clase.
- Crear un método static, llamado getInstance, que la primera vez que se llame cree la instancia y la almacene en la variable instance, y en las próximas llamadas simplemente se devuelva la variable que contiene la instancia.
- El constructor es privado-

Ejemplo:

- Ventana principal de un programa que solo se pueda instanciar una vez.



Adaptador (Adapter)

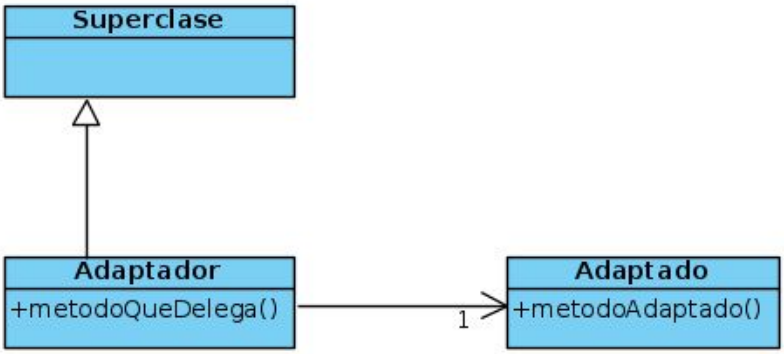
Se trata de un patrón estructural que acomoda la interfaz de una clase en otra que espera el cliente. Consiste en utilizar una clase sin incorporarla en la jerarquía (no se desea o no se puede usar herencia). Los métodos de la clase reutilizada no tienen que tener los mismos argumentos que los que deseamos necesariamente.

Cómo utilizarlo:

- Se crea una clase Adaptador que se incorpora en la jerarquía de herencia.
- La clase Adaptador está conectada mediante una asociación a la clase reutilizada (Adaptado).
- Los métodos de la clase Adaptador delegan en los de la clase Adaptado (no tienen que tener el mismo nombre ni argumentos necesariamente).
- Las clases que utilizan el Adaptador no saben que el Adaptador está delegando en otra clase.

Ejemplo:

- Una figura3D concreta necesita calcular su volumen, para ello ya existe una clase capaz de hacerlo.


Superclase
Adaptador
+metodoQueDelega()
1
Adaptado
+metodoAdaptado()

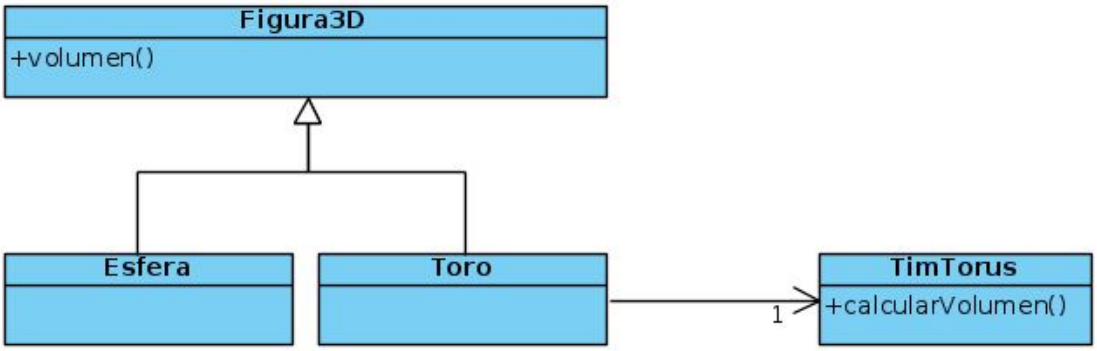
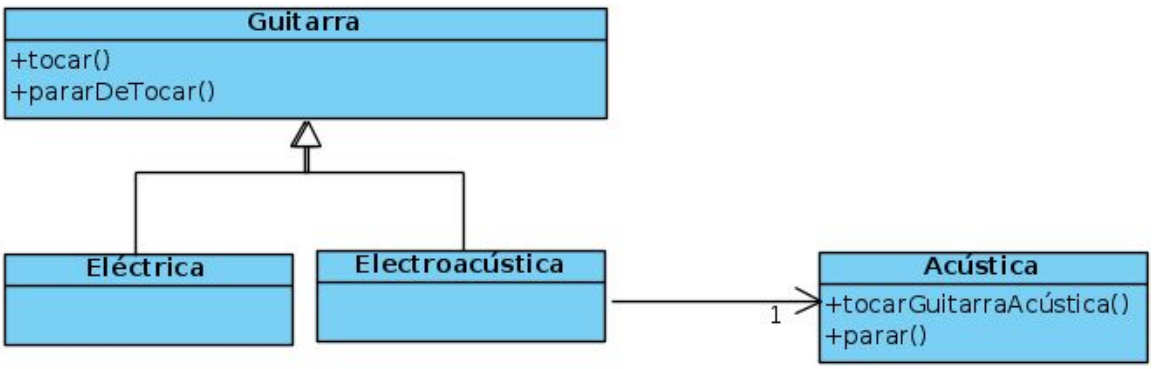


Figura3D
+volumen()
Esfera
Toro
1
TimTorus
+calcularVolumen()

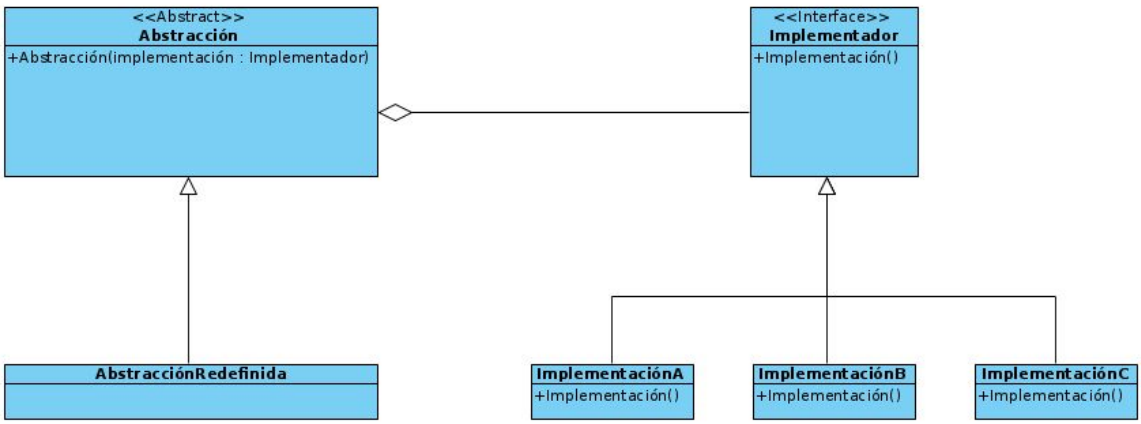

Guitarra
+tocar()
+pararDeTocar()
Eléctrica
Electroacústica
1
Acústica
+tocarGuitarraAcústica()
+parar()

Puente (Bridge) (Handle/Body)

Se trata de un patrón estructural que tiene el objetivo de desacoplar una abstracción de la implementación para que ambas puedan variar independientemente.
La idea es que la implementación sea un objeto que se asigna como una referencia.

Cómo utilizarlo:

- Crear una interfaz con el método del que se van a crear distintas implementaciones. Puede denominarse Implementador.
- La interfaz anterior es implementada tantas veces como implementaciones se deseen.
- Crear una clase abstracta que con contenga el método desacoplado. Además de un atributo de tipo Implementador y un método para asignarla.
- La clase anterior es implementada por una o varias clases que utilizan este patrón para tener un método desacoplado.

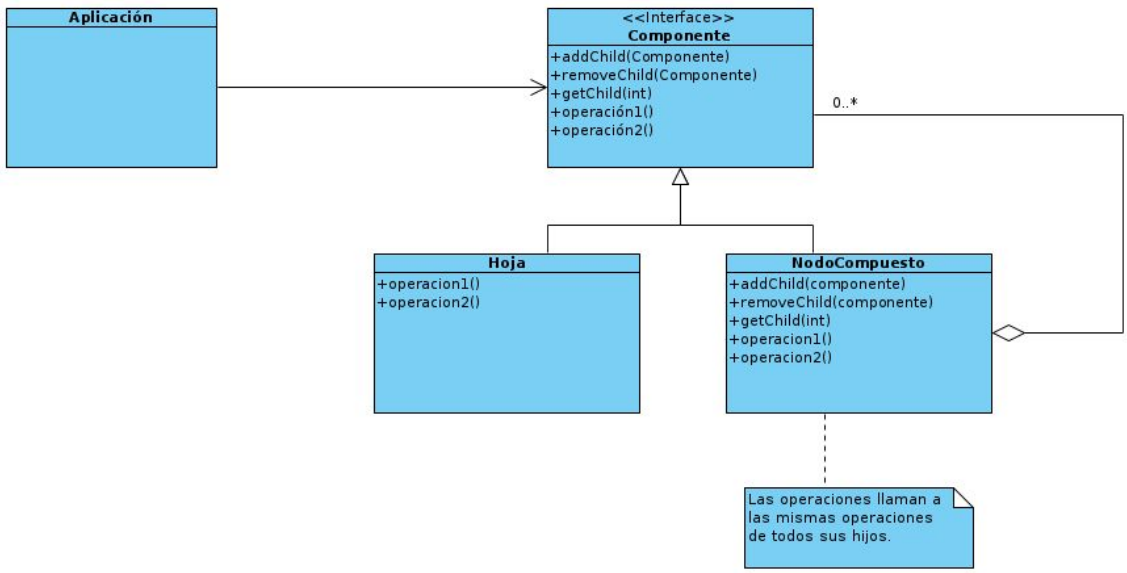


Composite

Este patrón tiene el objetivo de que un grupo de objetos pueda manejarse como un solo objeto. Simula la estructura de un árbol, permitiendo nodos hoja y nodos no terminales, estos últimos con la idea de que puedan tratarse junto a sus hijos como un todo. Esto se logra haciendo que las operaciones de un padre también se hagan sobre los hijos (y así recursivamente).
Los componentes pueden tener múltiples padres e hijos.

Cómo utilizarlo:

- Crear una interfaz Componente con:
  - Método para agregar hijos (recibe una referencia de otro componente).
  - Método para eliminar hijos (recibe una referencia de otro componente).
  - Método para obtener un hijo concreto (puede recibir un índice).
  - Una serie de operaciones que se realizan recursivamente hacia abajo.
- La interfaz anterior es implementada por:
  - Un nodo hoja, el cual no puede agregar, ni borrar, ni obtener hijos. Pero sí debe tener las operaciones.
  - Un nodo compuesto, que puede agregar varios hijos, eliminarlos y obtenerlos. Y debe implementar la operación, la cual llama a la misma operación de todos sus nodos hijos.



Proxy

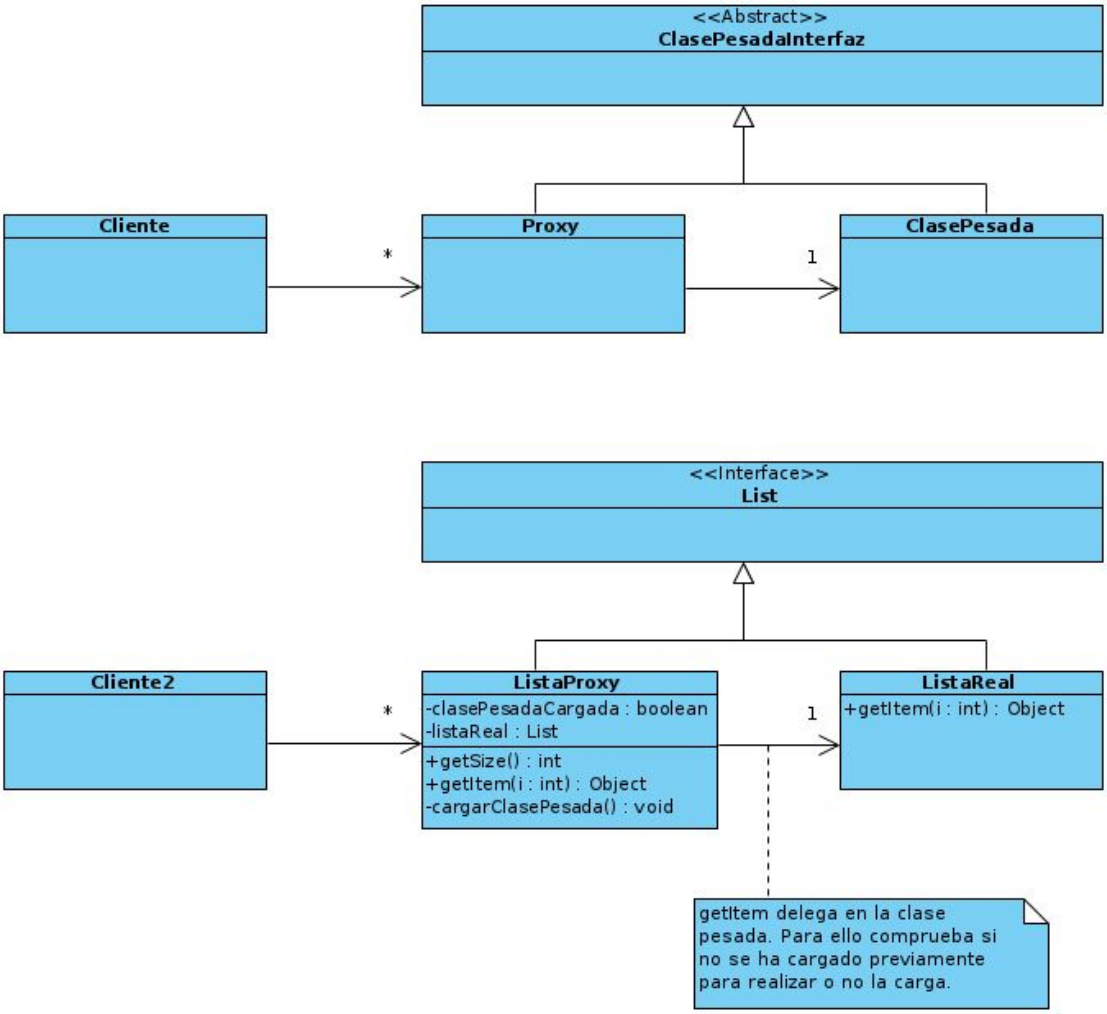
Se trata de un patrón estructural. Este patrón nace con la idea de acceder a clases pesadas. Clase pesada es como se le denomina a una clase remota.
El problema que soluciona es debido a que cargar un objeto pesado solicitado por el usuario puede hacer más lento el programa, por ello se pretende reducir esta carga a cuando sea estrictamente necesario sin reducir la funcionalidad. Es necesario que todos esos objetos estén disponibles en la memoria, aunque no estén necesariamente cargados.
La idea de este patrón es crear una versión más simple de la clase pesada, la cual pueda responder a las operaciones sencillas y evite la carga de la clase pesada en notables ocasiones.
Cargar toda la base de datos al principio del programa, o ir cargando objetos progresivamente a lo largo del mismo, son malas soluciones.

Cómo utilizarlo:

- Crear una versión más simple de la clase pesada, de forma que se pueda utilizar de igual modo que la clase pesada. Se le denomina Proxy.
- La clase Proxy delega a la clase pesada las operaciones que sean necesarias, realizando la costosa tarea de obtener la clase pesada. La clase pesada solo se carga una vez, en caso de ser necesaria.
- La clase Proxy puede tener operaciones sencillas implementadas, y que solo delegue en la clase pesada para ciertos métodos.
- La clase pesada y la clase Proxy comparten la misma interfaz.

Ejemplo:

- Una variable lista tiene un objeto de la tipo Lista.
- La variable realmente tiene un ListaProxy, evitando cargar la lista completa de objetos.
- ListaProxy carga la lista real cuando se quiere acceder a los objetos de la lista.
- ListaProxy es capaz de responder a operaciones sencillas como obtener el número de elementos, sin cargar la lista real.

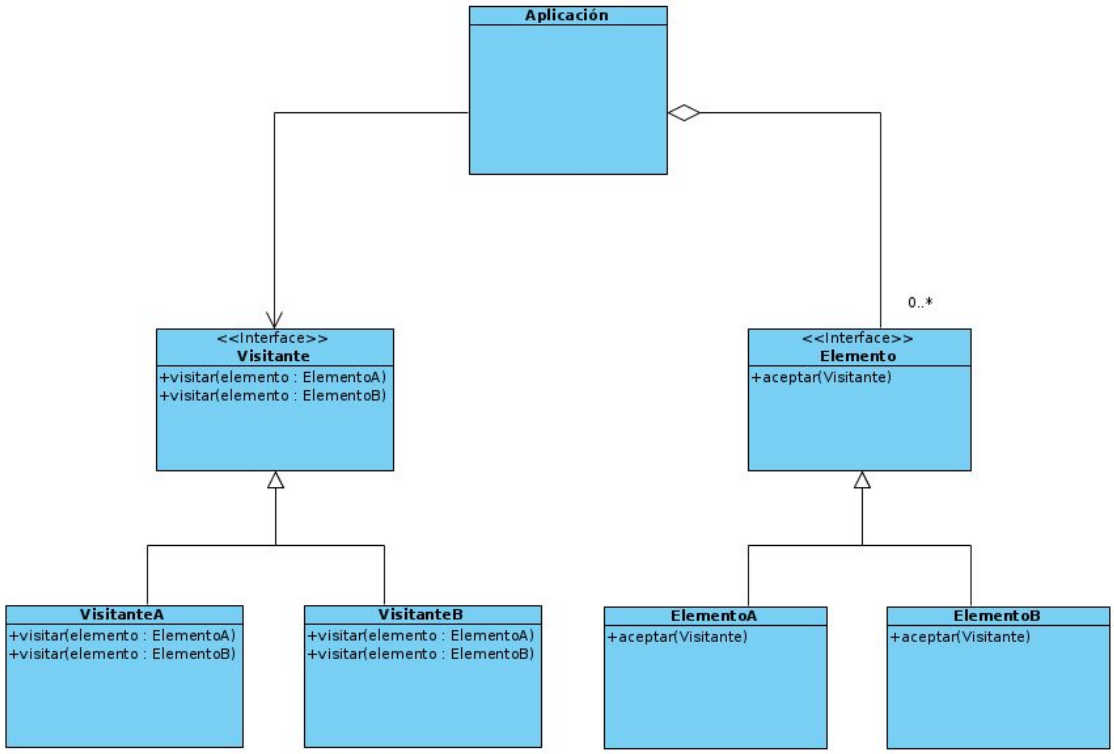

<<Abstract>>
ClasePesadaInterfaz
Cliente
Proxy
ClasePesada
*
1
<<Interface>>
List
Cliente2
ListaProxy
-clasePesadaCargada : boolean
-listaReal : List
+getSize() : int
+getItem(i : int) : Object
-cargarClasePesada() : void
ListaReal
+getItem(i : int) : Object
*
1
getItem delega en la clase pesada. Para ello comprueba si no se ha cargado previamente para realizar o no la carga.

Visitante (Visitor)

Es un patrón comportamental que tiene el objetivo de permitir a las aplicaciones realizar operaciones sobre una serie de instancias manteniendo el bajo acoplamiento y la separación. Para ello se utilizan los conceptos de visitar y aceptar. Una clase puede visitar a las instancias de un elemento para realizar cálculos en su interior.
Para visitar a una clase, la clase visitable cuenta con un método aceptar, éste método recibe la instancia de un visitante. El método aceptar envía una instancia de sí misma al método visitar del visitante, de este modo el visitante tiene acceso a la instancia concreta y puede realizar una serie de cálculos con los datos de ésta. Un visitante puede visitar a distintos elementos, visitantes distintos pueden hacer cálculos distintos.

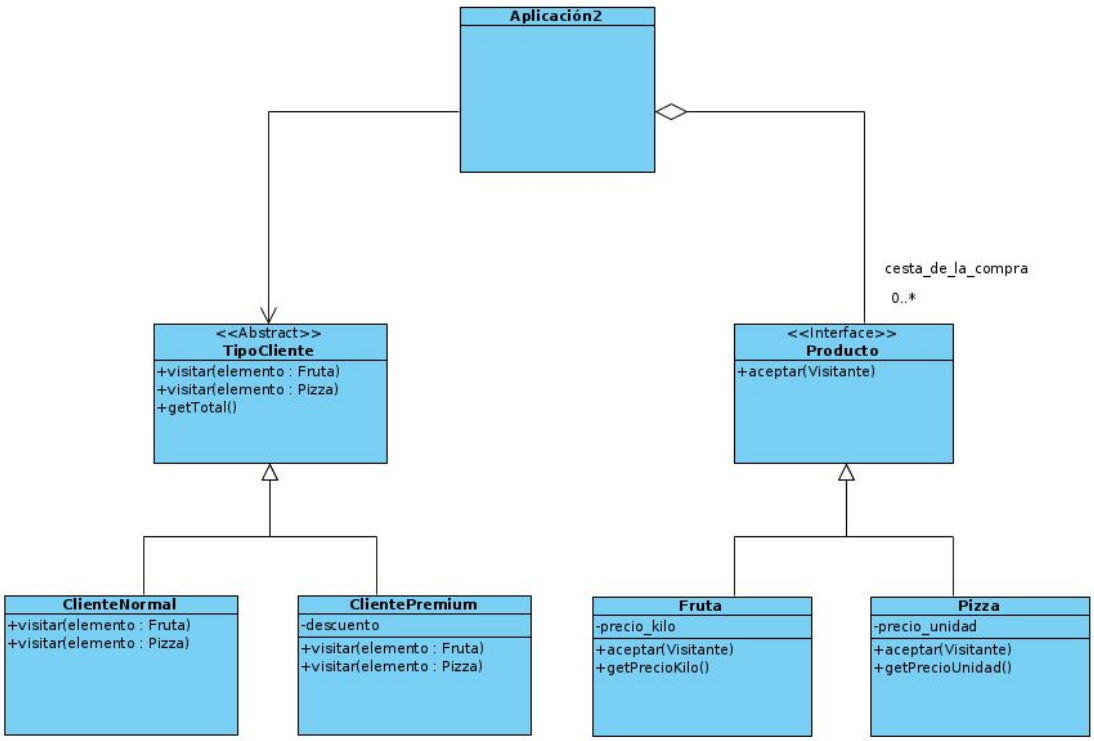
Cómo utilizarlo:

- Crear una interfaz Elemento, con el método aceptar(Visitante). Este método se utilizará para aceptar a un visitante y dar lugar a que el visitante realice su trabajo.
- Elemento puede ser implementada por los distintos tipos de elementos que aceptarán visitantes.
- Crear una interfaz Visitante con el método Visitar(Elemento). Puede sobrecargarse para crear métodos distintos de visita en función del tipo de Elemento, o incluso crear varios métodos distintos.
- Implementar la interfaz Visitante. Cada implementación puede realizar unos cálculos distintos a partir de los elementos visitados.



Ejemplo:

- Un supermercado con distintos producto y una tarifa para clientes premium.
- Crear una interfaz llamada Producto, con el método Aceptar(TipoCliente).
- Implementar la interfaz Producto con tantas subclases como productos se vendan, por ejemplo:
    - Fruta, en el cual establecemos el precio por kilo.
    - Pizza, donde establecemos el precio por unidad.
- Crear una interfaz TipoCliente, con el método visitar sobrecargado, ya que el precio se calcula de forma distinta en ambos:
    - visitar(Fruta)
    - visitar(Pizza)
- Crear las implementacioes de TipoCliente:
    - ClienteNormal: Obtiene la suma del precio de todos los productos.
    - ClientePremium: Obtiene la suma del precio de todos los productos, con un descuento del 5%.
- En la aplicación se crean tantas instancias de Producto como productos se estén comprando. Se llama al método aceptar de todos los productos indicando una referencia al tipo de cliente, el método visitar de TipoCliente va realizando el cálculo del precio en la instancia de TipoCliente, para luego poder obtenerlo y mostrarlo como el precio final de todos los productos que se desean obtener.



Abstracción-Caso (Abstraction-Occurrence)

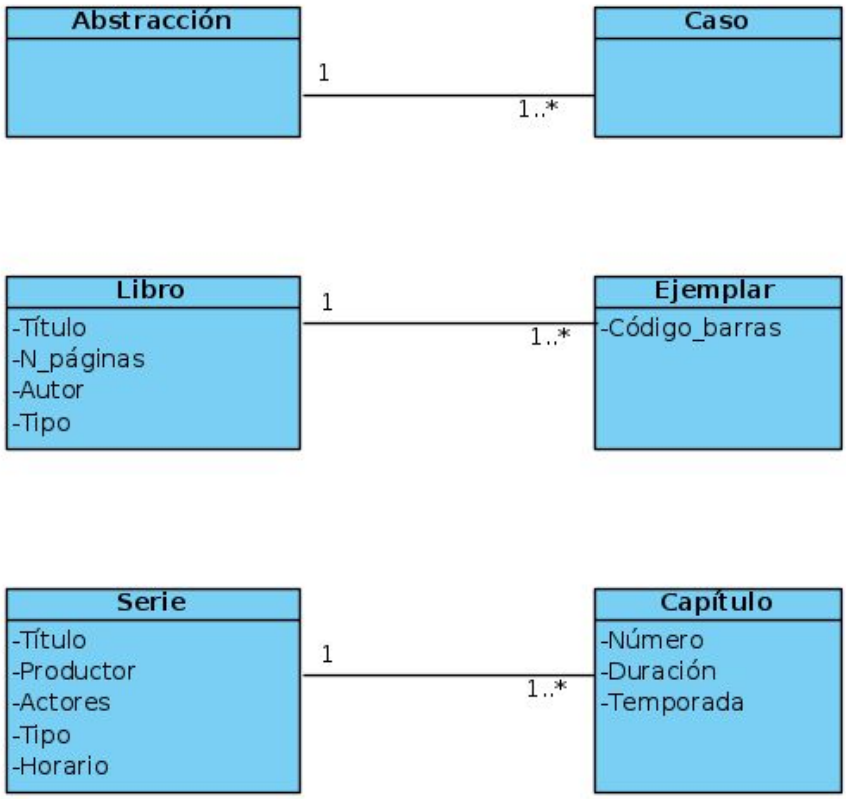
Aparecen objetos llamados casos, los cuales comparten información común pero se diferencian entre ellos de manera importante.
Se intenta evitar duplicar información evitando inconsistencias cuando se cambia la información en alguno de los objetos pero no en otros.
Trata de evitarse tener una sola clase con toda la información (ya que se repetiría y no se podría cambiar información sin editar todas). También se evita crear una clase distinta para cada caso; o utilizar herencia, ya que que los atributos son heredados pero no los valores, y por tanto habría que asignarlos en cada caso.

Cómo utilizarlo:

- Crear una clase Abstracción que contenga los datos comunes.
- Crear una clase Caso que represente las diferencias (los posibles casos).
- Conectar ambas clases mediante una asociación uno-a-muchos.

Ejemplo:

- Un catálogo de libros, para cada libro hay varios ejemplares.
- Cada ejemplar tiene un código de barras, pero comparte título, autor, número de páginas, etc.
- Abstracción:
    - Título, autor, número de páginas, etc.
- Caso:
    - Código de barras.


Abstracción
Caso
1
1..*
Libro
-Título
-N_páginas
-Autor
-Tipo
1
1..*
Ejemplar
-Código_barras
Serie
-Título
-Productor
-Actores
-Tipo
-Horario
1
1..*
Capítulo
-Número
-Duración
-Temporada

Jerarquía General (General Hierarchy)

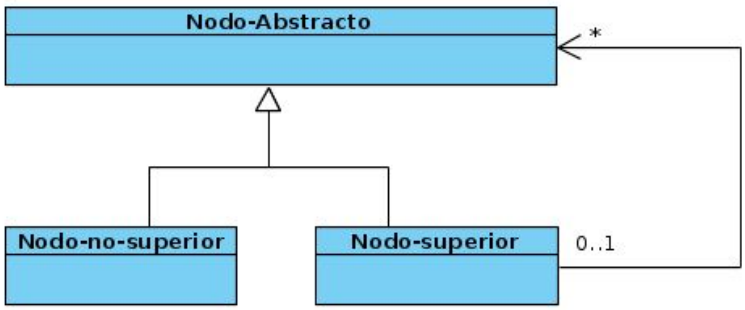
Se utiliza cuando objetos poseen una relación jerárquica natural, los cuales comparten características comunes.
Cada objeto de una jerarquía puede tener uno o más objetos encima de él (superiores) y cero o más debajo (subordinados). Se pretende impedir a algunos objetos tener subordinados.

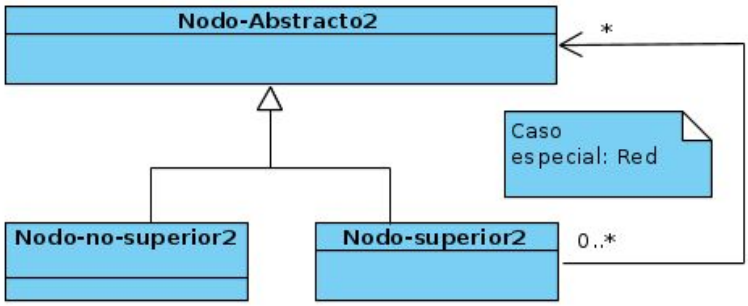
Cómo utilizarlo:

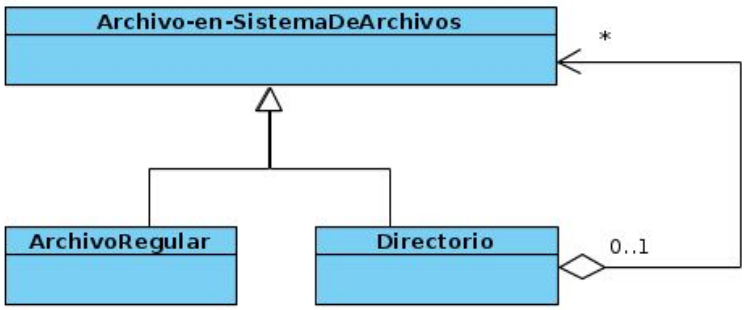
- Crear una clase Nodo-abstracto que representa las características que poseen los objetos de la jerarquía. Una de las características obligatorias de la jerarquía es que los nodos pueden tener nodos superiores.
- Crear, al menos, dos subclases de Nodo-abstracto.
    - Nodo-superior: Conectada con otra asociación a la clase Nodo-abstracto, de forma que pueda tener subordinados de tipo Nodo-superior o Nodo-no-superior. Esta asociación puede ser uno-a-muchos o muchos-a-muchos (una red, un nodo puede tener muchos superiores).
    - Nodo-no-superior

Ejemplos:

- Un sistema de archivos con dos tipos de archivos:
    - Archivos regulares.
    - Archivos directorio: pueden contener más archivos regulares o directorios.
- Interfaz de usuario:
    - Botones, etiquetas, etc.
    - Ventanas y paneles: pueden contener otros objetos en su interior.


Nodo-Abstracto
*
Nodo-no-superior
Nodo-superior
0..1


Nodo-Abstracto2
*
Caso
especial: Red
Nodo-no-superior2
Nodo-superior2
0..*


Archivo-en-SistemaDeArchivos
*
ArchivoRegular
Directorio
0..1

Actor-Rol (Player-Role)

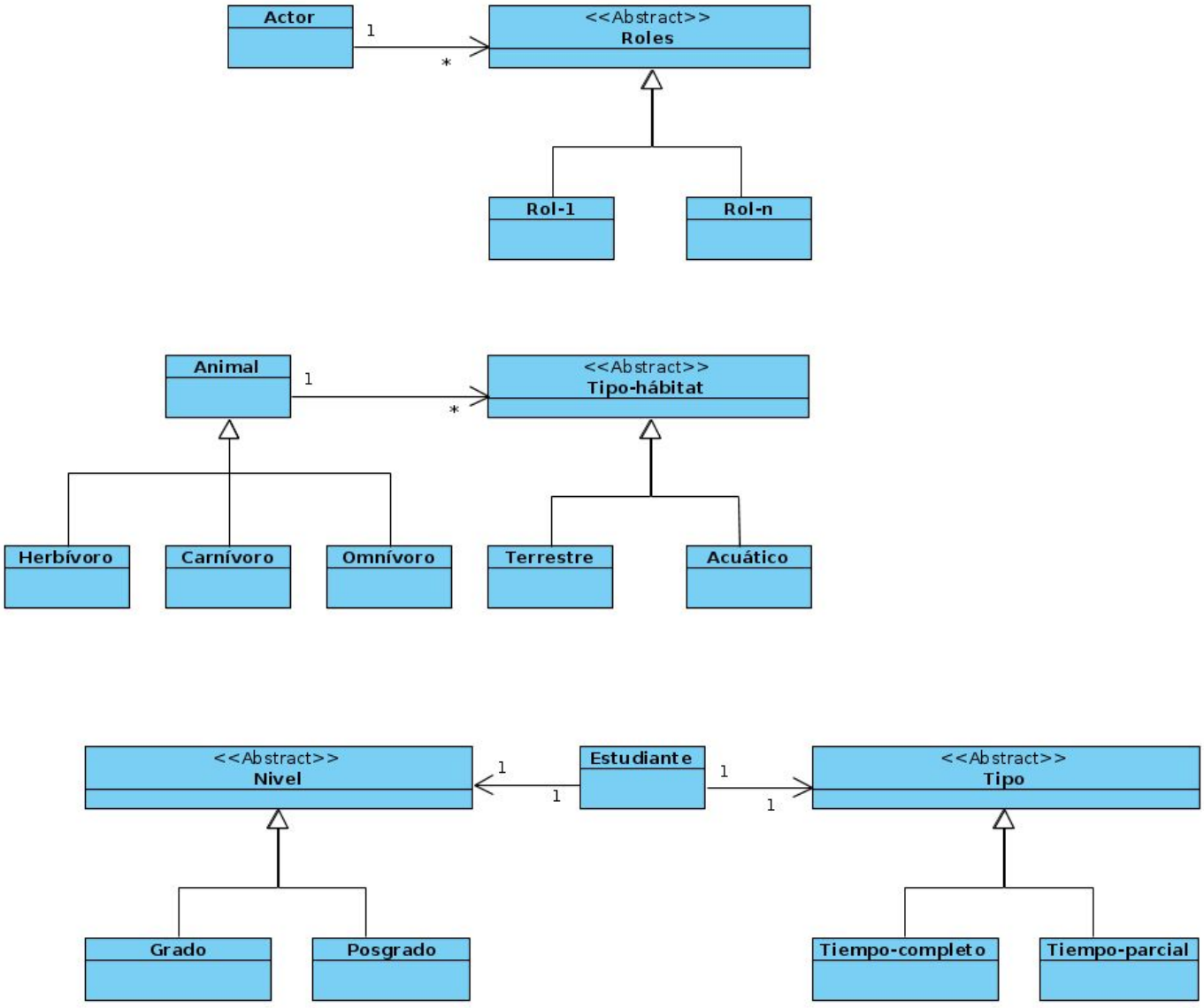
Un actor puede tener distintos roles, según el contexto concreto. Un rol es una serie de características concretas asociadas a un actor.
A un actor se le pueden asignar múltiples roles, y puede cambiar de rol.
Se pretende conseguir esto sin usar herencia múltiple y sin permitir que una instancia pueda cambiar de clase. Tampoco es válido mezclar todas las características en una clase en lugar de usar roles (clases excesivamente grandes y poco dinámicas). Los roles no pueden ser subclases de actor.

Cómo utilizarlo:

- Crear una clase Actor.
- Crear una clase abstracta Roles, la cual será superclase de los distintos roles. Podría ser una interfaz en lugar de una clase abstracta, pero entonces los roles no podrán acceder a la información del Actor.
- Asociar Actor con la clase abstracta Roles, puede ser de multiplicidad uno-a-uno o uno-a-muchos.

Ejemplos:

- Un animal puede tener tres roles: terrestre, acuático o ambos. Algunos animales pueden cambiar de rol. Un animal puede ser de tipo carnívoro, herbívoro, o ambos. Esto no es un Rol porque no puede cambiar nunca, los actores pueden cambiar de roles.
- Un estudiante puede tener 2 roles:
    - Ser estudiante a tiempo completo o parcial.
    - Ser estudiante de grado o de posgrado.

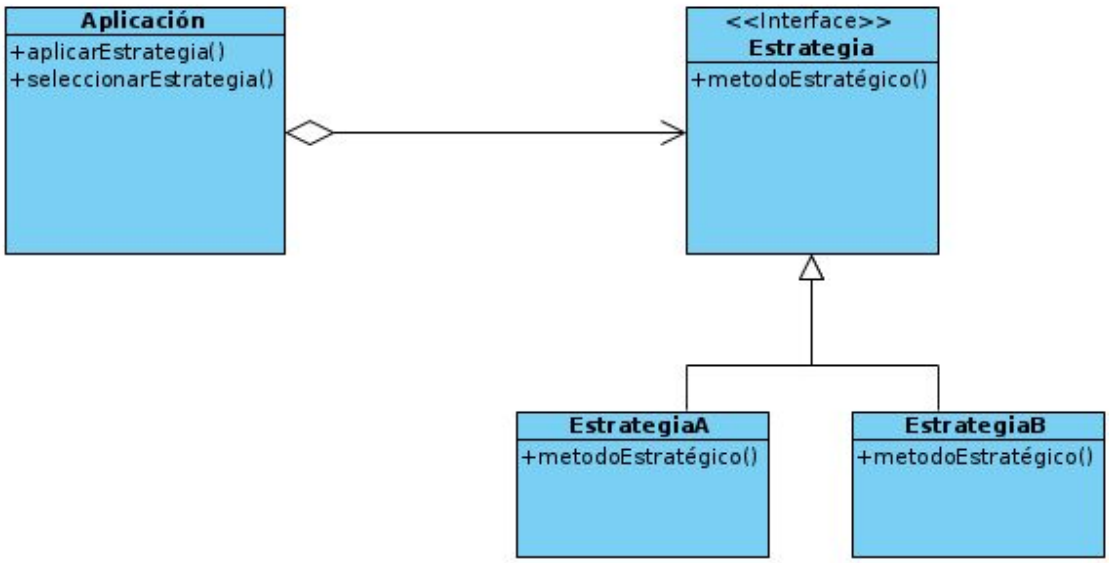

Actor
1
*
<<Abstract>>
Roles
Rol-1
Rol-n
Animal
1
*
<<Abstract>>
Tipo-hábitat
Herbívoro
Carnívoro
Omnívoro
Terrestre
Acuático
<<Abstract>>
Nivel
1
1
Estudiante
1
1
<<Abstract>>
Tipo
Grado
Posgrado
Tiempo-completo
Tiempo-parcial

Strategy

Este patrón permite crear distintas estrategias (implementaciones) e intercambiar o decidir cuál se usa en función de la necesidad

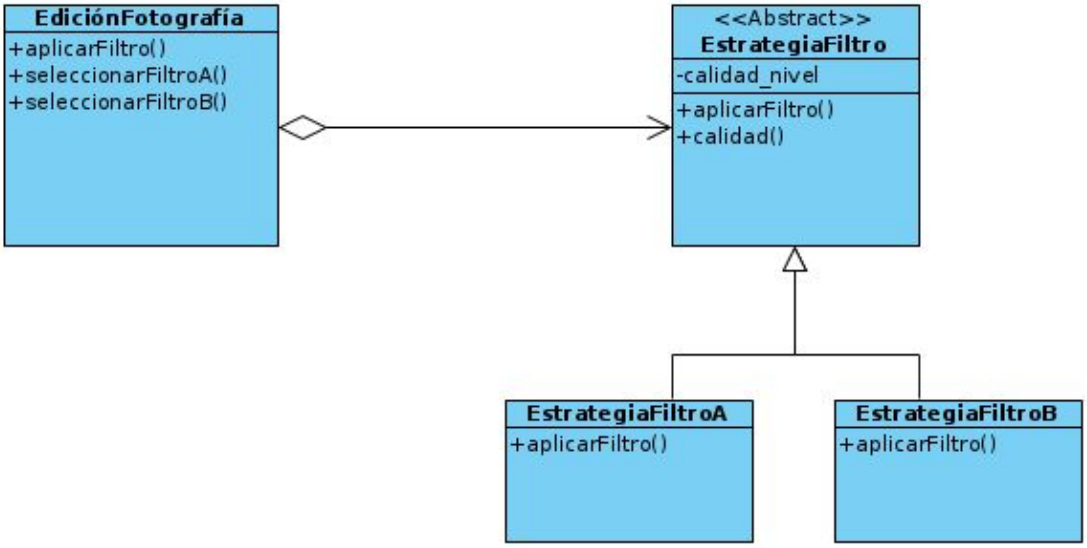
Cómo utilizarlo:

- Crear una interfaz Estrategia con un método que será redefinido, la interfaz puede tener otros métodos o atributos.
- Extender la interfaz creando estrategias en las que se redefine el método tantas veces como necesitemos.
- Una vez hecho esto, desde una clase cliente podemos tener un método que realiza una acción utilizando una instancia de tipo Estrategia. De este modo podemos instancia tipos concretos de estrategia guardándolos en un atributo, decidiendo así cuál usar en cada momento.



Ejemplo:

- Un programa de edición de fotografía puede usar dos tipos de filtros, en función del que el usuario elija.
- Crear la interfaz EstrategiaFiltro con el método aplicarFiltro.
- Extender la interfaz anterior con los métodos aplicarFiltroA y aplicarFiltroB.
- Crear el programa principal, en el cual se incluye un atributo de tipo EstrategiaFiltro. Además de dos métodos llamados seleccionFiltroA y seleccionFiltroB. El primero crea una instancia de EstrategiaFiltroA y la almacena en el atributo, y el segundo igual pero para EstrategiaFiltroB. Crear otro método llamado aplicarFiltro que llama al método aplicarFiltro de la estrategia indicada en el atributo.

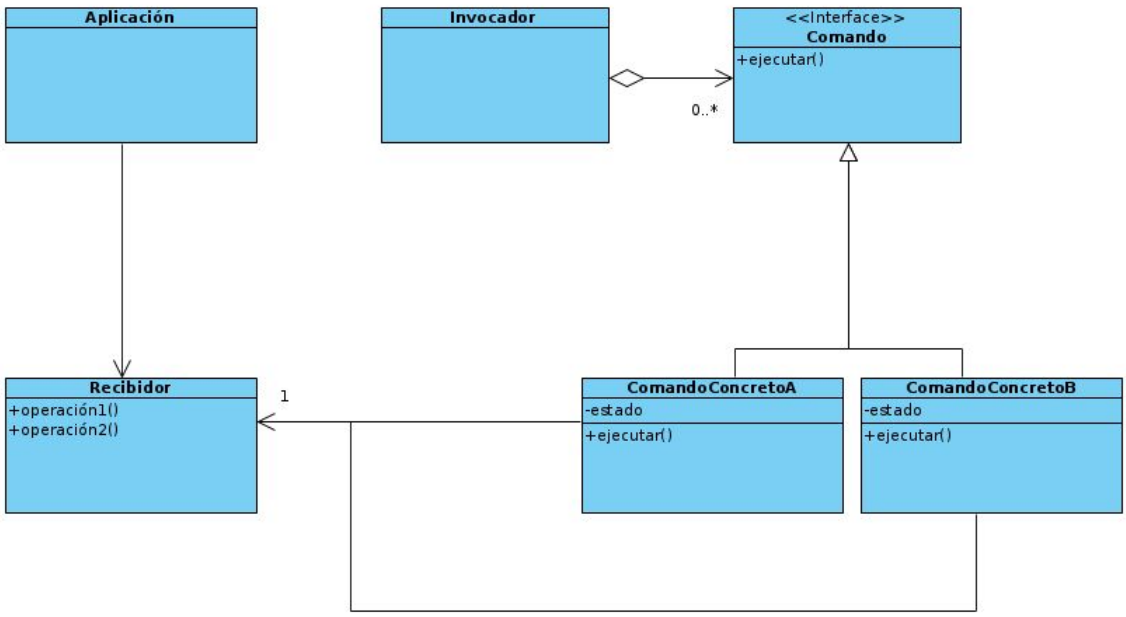

EdiciónFotografía
+aplicarFiltro()
+seleccionarFiltroA()
+seleccionarFiltroB()
<<Abstract>>
EstrategiaFiltro
-calidad_nivel
+aplicarFiltro()
+calidad()
EstrategiaFiltroA
+aplicarFiltro()
EstrategiaFiltroB
+aplicarFiltro()

Comando (Command)

Este patrón permite que las solicitudes de los clientes se convierten en objetos, permitiendo generalizar a los clientes dependiendo sólo de su tipo de solicitud, encolar o registrar solicitudes y convertir en reversibles las operaciones.
En lugar de ejecutar una operación, ejecutar un comando que ejcute la operación.

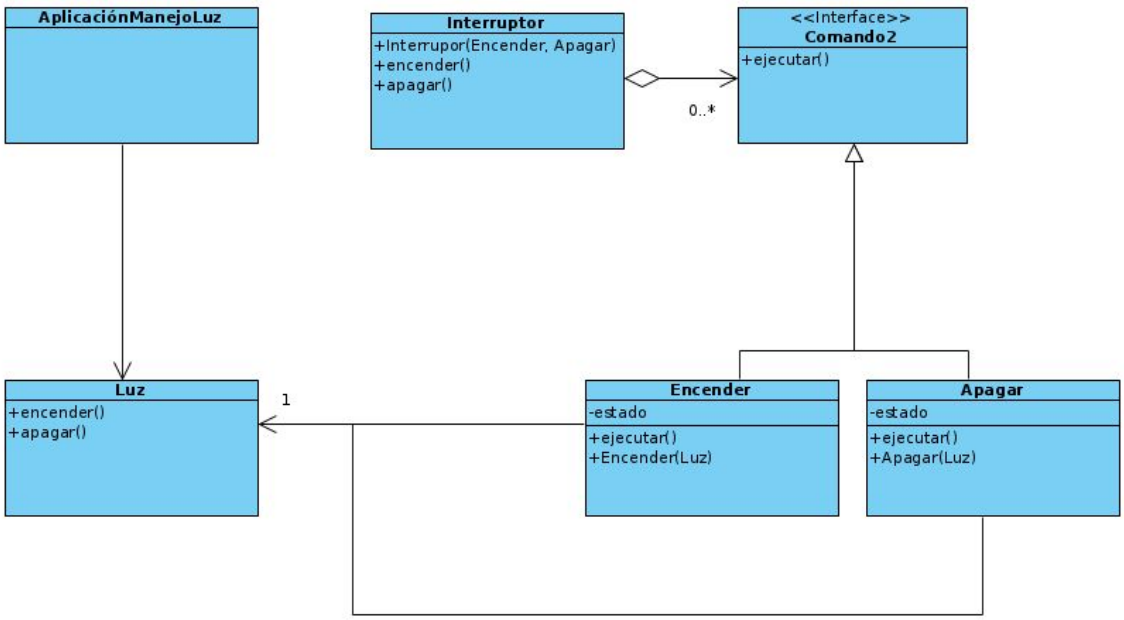
Cómo utilizarlo:

- Crear una clase Invocador. El objetivo de es ejecutar los comando, contiene los comando y los utiliza según la petición del usuario.
    - Podría recibir los posibles comandos en el constructor, por ejemplo.
    - El usuario llama a un método del invocador, y éste al método ejecutar del comando adecuado.
- Crear una interfaz Comando con el método ejeciutar(). Este método es el encargado de llevar a cabo la acción.
- Implementar la interfaz anterior tantas veces como comandos se requieran. Es posible almacenar un estado. Hay que redefinir el método ejecutar para que efectúe una acción sobre Recibidor, para ello necesitaremos enviarle una referencia de la instancia (podría hacerse en el constructor).
- Recibidor es el objeto que recibe la acción del comando.
- En el cliente podríamos crear los comandos, pasándole la referencia del recibidor; y a continuación crear el Invocador, pasándole las referencias de los comandos. Para usarlo habría que pedirle al invocador que haga una acción u otra mediante sus métodos y a través de los comandos que le hemos entregado.



Ejemplo:

- Un interruptor de la luz, el cual tiene dos comando: encendido y apagado.
- El recibidor es una luz, tiene dos métodos:
    - encender().
    - apagar().
- Creamos la interfaz comando con el método ejecutar.
- Creamos las dos implementaciones de Comando:
    - Encender: llama al método endender() de la luz.
    - Apagar: llama al método apagar() de la luz.
- El invicador será el interruptor, creamos la clase Interruptor con dos referencias a Comando, y dos métodos. Uno de los métodos llama al ejecutar del comando Encender, e el otro de la misma manera para Apagar.
- En el programa cliente creamos las dos instancias de Comando: Encender y Apagar; a las cuales le damos una referencia de la instancia de Luz. Y por último creamos una instancia de Interruptor, indicándole las dos instancias anteriores. Con todo esto ya tendríamos un interruptor capaz de encender y apagar la luz por medio de comandos.



Observador (Observer)

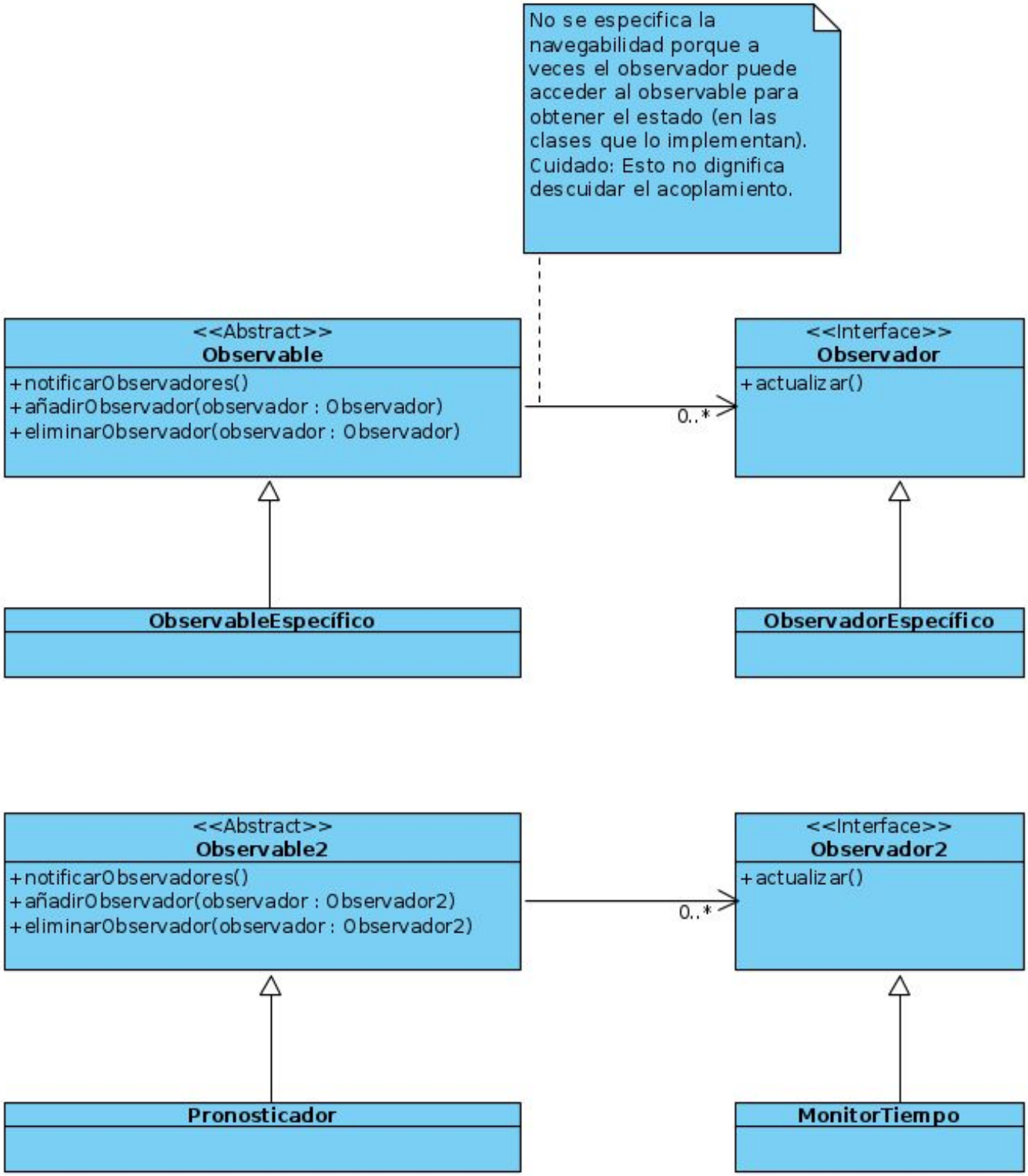
Este patrón sirve para permitir que un objeto pueda comunicarse con otros sin conocer las clases a las que pertenecen. Se trata de conectar una clase con otras de un modo flexible y reduciendo el acoplamiento, de forma que si se producen cambios en una, se pueda alertar a las otras y que se produzca una reacción en ellas.
Hay que evitar que una clase contenga referencias a la otra, ya que esto produce acoplamiento. También hay que evitar recurrir a la herencia múltiple.

Cómo utilizarlo:

- Crear una interfaz llamada Observable.
- Crear una interfaz llamada Observador.
- Crear una colección de instancias de la clase Observador dentro de la clase Observable.
- Crear métodos para añadir y eliminar observadores del observable. También un método para notificar a los observadores.
- La clase Observador solo tiene el método actualizar, el cual es llamado desde el observable cuando se llama a notificar observadores.
- De este modo podemos tener objetos de tipo Observable que tienen una lista de otros objetos de tipo Observador, los cuales han sido añadidos por un método de forma dinámica y también pueden ser eliminados. De esta forma el objeto Observable llama a noticarObservadores() cuando se produzca un cambio, y este llama a actualizar() de todos los observadores registrados.

Ejemplos:

- Servicio de predicción del tiempo:
    - Pronosticador: Hace predicciones meteorológicas, sería el objeto observable.
    - El objeto observable notifica a todos los interesados cuando se realiza una predicción
- Es la base del modelo MVC.


No se especifica la navegabilidad porque a veces el observador puede acceder al observable para obtener el estado (en las clases que lo implementan). Cuidado: Esto no dignifica descuidar el acoplamiento.
<<Abstract>>
Observable
+notificarObservadores()
+añadirObservador(observador : Observador)
+eliminarObservador(observador : Observador)
0..*
<<Interface>>
Observador
+actualizar()
ObservableEspecífico
ObservadorEspecífico
<<Abstract>>
Observable2
+notificarObservadores()
+añadirObservador(observador : Observador2)
+eliminarObservador(observador : Observador2)
0..*
<<Interface>>
Observador2
+actualizar()
Pronosticador
MonitorTiempo

Delegación (Delegation)

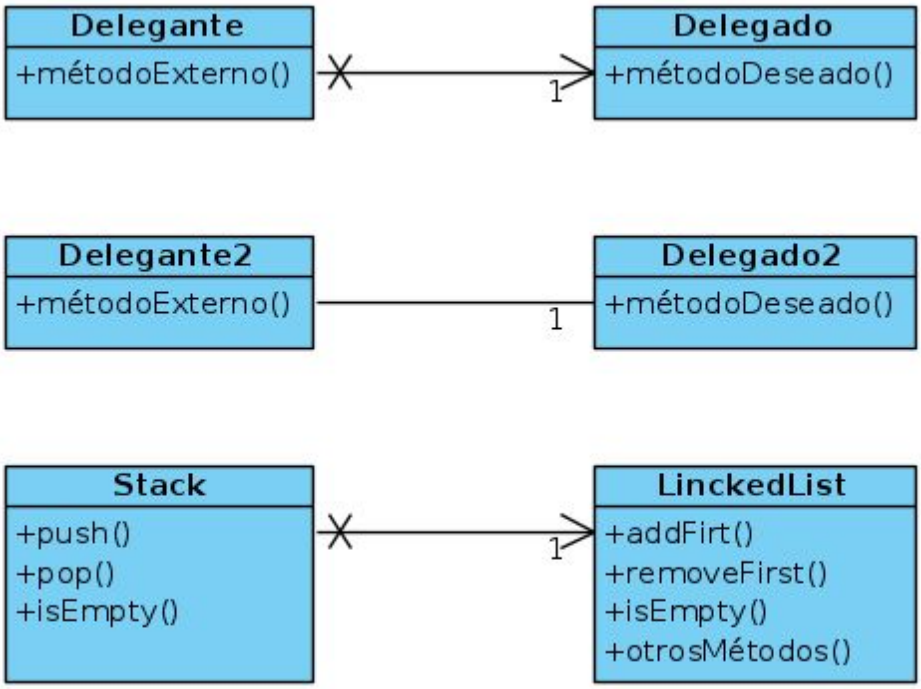
Se necesita una implementación determinada de una operación que ya tiene implementada otra clase, pero para heredar la operación no conviene convertir a la primera subclase de la segunda ni tampoco utilizar herencia. Tampoco se quiere repetir código ni provocar que una operación la haga una clase que no es adecuada para ello.
Las ligaduras son peligrosas, no hay que abusar de este patrón.

Cómo utilizarlo:

- La clase que desea acceder a un método de otra clase se denomina Delegante.
- La clase en la que se delega se denomina Delegado.
- Crear un método en la clase Delegante que solo llame al método deseado de la clase Delegado. Esto provoca una asociación entre las dos clases, algo bastante delicado porque aumenta la complejidad, por ello hay que evitar usar este patrón si ésta asociación no existe previamente o no se puede crear dinámicamente. Esta asociación de direccional del Delegante al Delegado, o bidireccional.

Ejemplo:

- Una clase Stack puede ser creada con facilidad a partir de una clase colección prexistente, como LinkedList:
    - El método push() de la clase Stack simplemente llamaría al método addFirst() de la clase LinkedList.
    - El método pop() llamaría al método removeFirst().
    - El método isEmpty() simplemente delegaría en el método del mismo nombre.
    - No se utilizarían los otros métodos de la clase LinkedList ya que no tienen ningún uso dentro de la clase Stack.


Delegante
+métodoExterno()
Delegado
+métodoDeseado()
1
Delegante2
+métodoExterno()
Delegado2
+métodoDeseado()
1
Stack
+push()
+pop()
+isEmpty()
LinckedList
+addFirt()
+removeFirst()
+isEmpty()
+otrosMétodos()
1

Fachada (Façade)

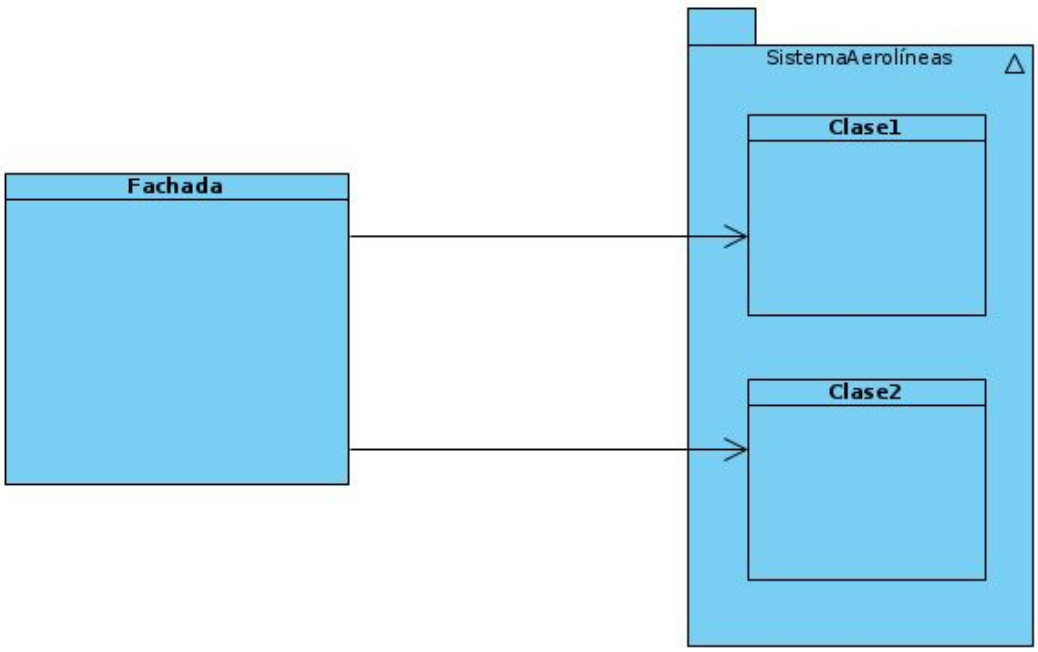
Se utiliza como medio para acceder a un paquete o clases de uso completo. Se trata de una clase intermediaria que facilita el uso del sistema o de una parte del mismo.

Cómo utilizarlo:

- Crear una clase Fachada.
- Crear métodos en la clase Fachada que sirvan para interactuar con el subsistema que deseamos simplificar.
- De este modo los subsistemas que interactúan con Fachada no necesitan comprender la dificultad del subsistema que están utilizando, y en caso de que se realicen cambios solo hay que modificar la clase fachada, en lugar de todos los subsistemas que hacen uso de este.

Ejemplo:

- Un sistema de gestión de aerolíneas es muy complejo, pero a la vez hay muchos subsistemas que necesitan interactuar con él, y estos no pueden arriesgarse a los cambios que pueda sufrir ese sistema.
- Para solucionar el problema anterior se crea una o varias fachadas que permitan una mejor interacción y seguridad ante los cambios.



Inmutable (Immutable)

Se pretende obtener objetos con la certeza de que no van a cambiar de contenido inesperadamente. No puede existir ninguna forma de modificar sin permiso un objeto.

Cómo utilizarlo:

- El constructor de la clase Inmutable puede ser el único lugar donde las variables son modificadas.
- No puede existir ningún método que cambie los valores de las variables
- Si un método necesita cambiar los valores, debe crear y devolver una nueva instancia de la clase Inmutable.

Ejemplo:

- Clase Punto, con valores x, y, z los cuales se desea que no puedan alterarse nunca.
- Solo existe el método traslación, el cual devuelve una nueva instancia de punto.

```
public final class Punto {
        private final float x;
        private final float y;
        private final float z;

        public Punto(float x, float y, float z) {
                this.x = x;
                this.y = y;
                this.z = z;
        }
        public Punto trasladar(float x, float y, float z) {
                return new Punto(float x, float y, float z);
        }
        ...
}
```


Punto
-X : float
-Y : float
-Z : float
+Punto(x : float, y : float, z : float)
+trasladar(x : float, y : float, z : float) : Punto

Interfaz Solo–Lectura (Read–Only Interface)

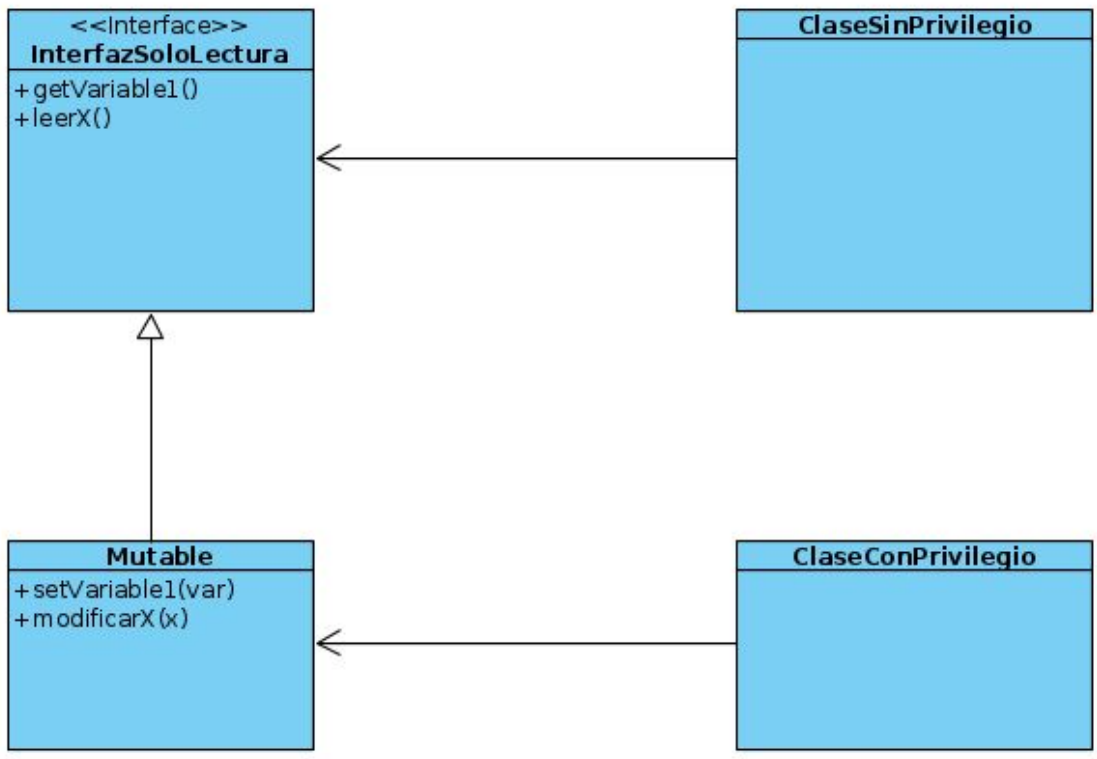
Es como el patrón Inmutable,, con la diferencia de que éste permite a ciertas clases privilegiadas modificar las instancias. Se trata de lograr que haya clases que vean a unos objetos como objetos de solo lectura, mientras que otras privilegiadas son capaces de alterar el estado de estas instancias.

Cómo utilizarlo:

- Crear una interfaz InterfazSoloLectura.
- Crear una clase Mutable que implementa la interfaz anterior.
- InterfazSoloLectura contendrá únicamente los métodos que no realizan cambios en los atributos de la clase Mutable.
- Las clases que realizan cambios recibirán instancias de Mutable.
- Las clases que no deben realizar cambios recibirán instancias de tipo InterfazSoloLectura, de modo que no conozcan la existencia de los métodos mutables.

Ejemplo:

- Se utiliza en MVC, de modo que nos asegura que la vista no puede modificar los datos del modelo que no deseemos.



Cadena de responsabilidad (Chain of responsability)

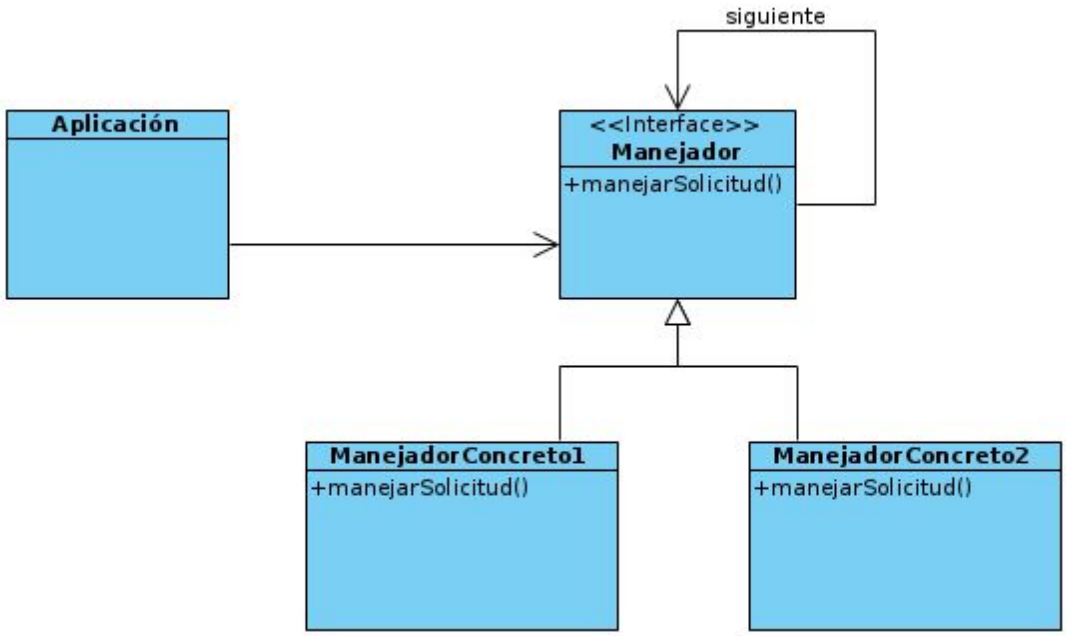
Consiste en una cadena de clases que responden a la petición de un cliente. Lo característico de este patrón, es que la cadena de clases se va pasando la petición hasta que una de ellas la responde.

Cómo utilizarlo:

- Crear una interfaz manejador, la cual contiene una referencia a una interfaz del mismo tipo.
- La interfaz manejador puede implementarse de tantas formas como se deseen.
- La idea es que cuando se llame a un método de un manejador, este llame al mismo método del manejador del cual tiene la referencia, hasta que un manejador pueda responder con una acción.

Ejemplo:

- El ejemplo más típico es una cadena de mando, donde uno tiene responsabilidades sobre otro, de modo que cada uno va avisando al siguiente con una orden hasta que la orden se puede realizar.

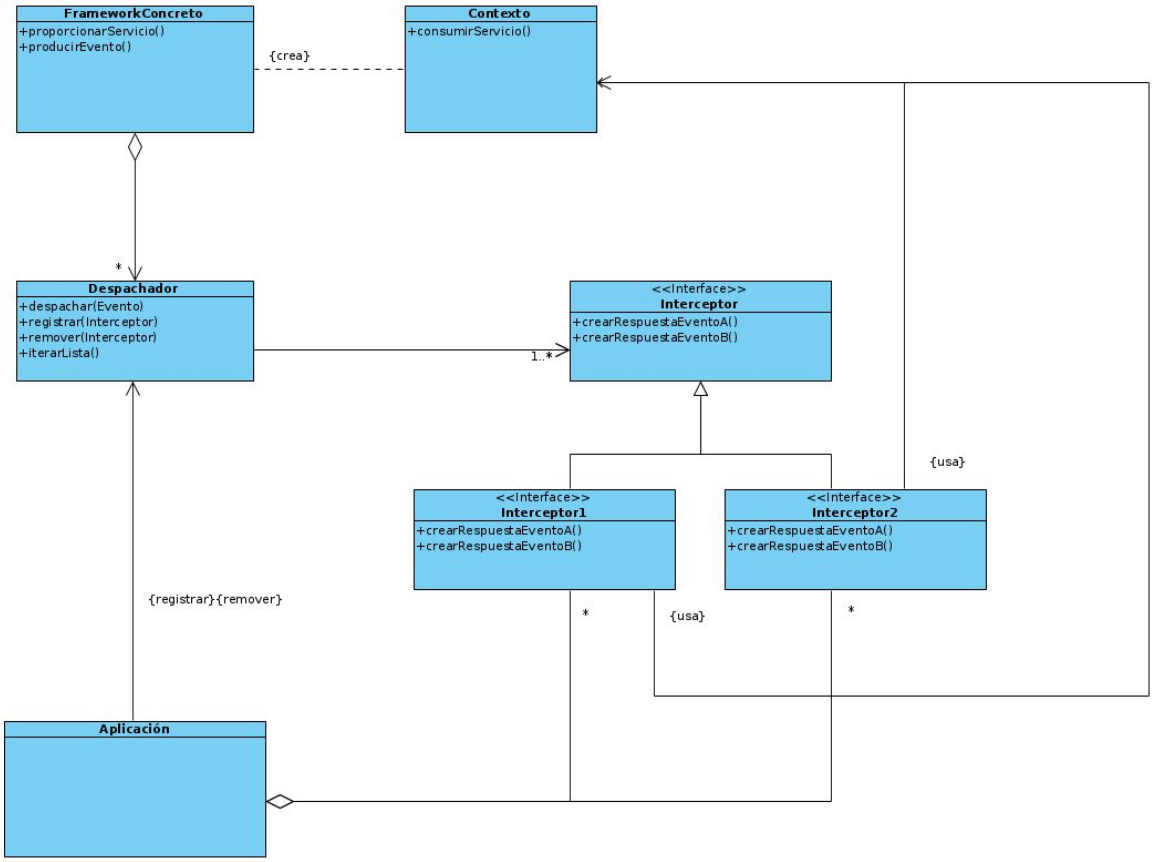


Interceptor

Es un patrón arquitectónico que permite añadir servicios de manera transparente a un marco de trabajo y ser disparados automáticamente cuando ocurren ciertos eventos. De este modo nos anticipamos a futuras demandas de servicios.

Cómo utilizarlo:

- Crear una interfaz Interceptor, la cual se implementará con el objetivo de proporcionar respuestas a ciertos eventos. Incluir los métodos necesarios que responderán a los eventos.
- Implementar la interfaz anterior, creando las respuestas específicas para cada evento que se desea responder.
- Crear una clase Despachador, la cual contiene una colección de interceptores, y métodos para registrarlos, borrarlos y recorrerlos. Cuenta también con un método Despachar, que es el encargado de recibir un evento y buscar entre los interceptores uno que lo responda.
- Desde la Aplicación se crean los interceptores y se registran en el Despachador.
- Crear una clase llamada FrameworkConcreto, la cual tiene acceso a los despachadores, esta clase crea un Contexto que es modificado y afectado por los servicios que se dispone, los cuales son las respuestas de los eventos que FrameworkConcreto pide resolver al Despachador. El Contexto solicita un servicio al FrameworkConcreto y este produce un evento que resuelve el Despachador.

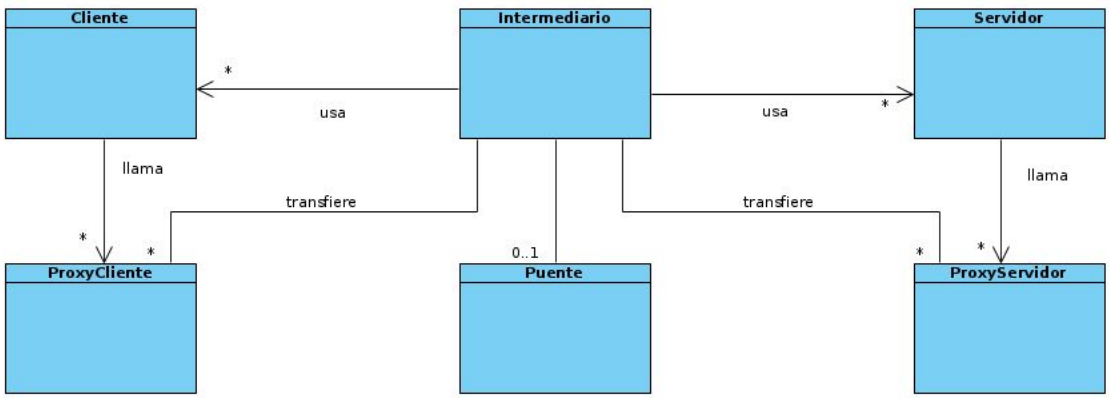

FrameworkConcreto
+proporcionarServicio()
+producirEvento()
{crea}
Contexto
+consumirServicio()
*
Despachador
+despachar(Evento)
+registrar(Interceptor)
+remover(Interceptor)
+iterarLista()
1..*
<<Interface>>
Interceptor
+crearRespuestaEventoA()
+crearRespuestaEventoB()
{usa}
<<Interface>>
Interceptor1
+crearRespuestaEventoA()
+crearRespuestaEventoB()
<<Interface>>
Interceptor2
+crearRespuestaEventoA()
+crearRespuestaEventoB()
{registrar}{remover}
*
{usa}
*
Aplicación

Broker

Este patrón se utiliza para modelar sistemas distribuidos compuestos de componentes totalmente desacoplados.
La idea consiste en tener una clase intermediaria entre el cliente y el servidor, de modo que sea esta intermediaria la que busca y decide a qué servidor conecta el cliente, además de poder buscar los servicios disponibles, etc. De este modo el cliente no está acoplado al servidor, el servidor puede variar y pueden añadirse nuevos servidores y servicios.
Es necesario empaquetar y desempaquetar datos, además de enviarlos y recibirlos; esto también se hace con clase específicas para ello, las cuales intervienen en todas las conexiones.

Cómo utilizarlo:

- Crear una clase Cliente, el cual cuenta con una referencia a un Proxy que le permite empaquetar y desempaquetar lo datos, además de enviar y responder solicitudes.
- El Cliente puede estar conectado a diversos proxys, en función de las distintas conexiones que esté estableciendo.
- El Servidor se implementa de un modo parecido al Cliente, cuenta con otro Proxy específico.
- Crear otra clase denominada Intermediaria, la cual es capaz de registrar los servicios que le dicta el servidor, de buscar entre los servicios disponibles, de buscar un servidor disponible, de reenviar solicitudes y respuestas, etc. Está directamente conectada con los clientes y servidores, por lo tanto también necesita empaquetar datos, desempaquetar,... Para ello utiliza una instancia de otra clase específica idéntica a los proxys, denominada Puente.



Reflection

Este patrón proporciona un mecanismo para cambiar el comportamiento de un software de forma dinámica. Permite modificar estructuras y llamadas a funciones. Gracias a este patrón el software es consciente de sí mismo y puede modificarse.

Está dividido en dos partes:

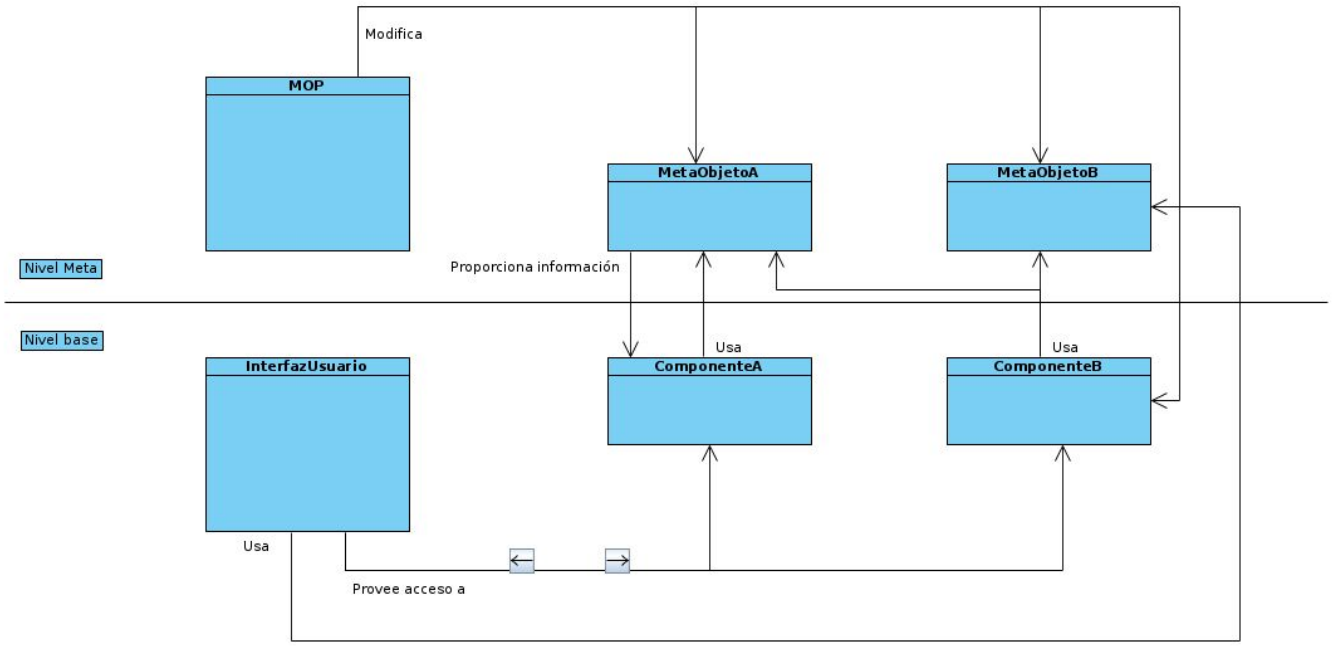
- Nivel Meta: Proporciona información sobre las propiedades del sistema y hace al software consciente de sí mismo. Consta de los meta-objetos, los cuales encapsulan y representan información del software (cabeceras, tipos, etc).
- Nivel Base: Consta de la lógica de la aplicación. Es la implementación del Nivel Meta. Implementa los meta-objetos.

Se utiliza el concepto MOP (Meta-Object Protocol): Se encuentra en el Nivel Meta, y se trata de una interfaz especializada para la administración y mantenimiento del sistema, permite configurar y modificar dinámicamente los meta-objetos. MOP recibe como parámetro las implementaciones que se hagan en el Nivel Meta y es el responsable de integrar los cambios. MOP puede tener clientes procedentes del nivel base, de un usuario privilegiado u otras aplicaciones. Se utiliza este concepto para poder realizar cambios en todo el sistema de forma segura, ya que es posible cambiar todas las conexiones e interacciones, MOP se asegura de la corrección de estos cambios.

Cómo utilizarlo:

- Implementar una clase MOP, capaz de decidir cómo se relacionan los componentes e implementan, asegurando que el sistema esté correcto.
- Definir los MetaObjetos, que proporcionan interfaces que luego se implementarán.
- Crear los componentes, los cuales utilizan los meta objetos para trabajar con el sistema.
- Mediante MOP, definir la configuración del sistema

El siguiente diagrama muestra la estructura del patrón, pero las relaciones que se producen son un ejemplo, esas relaciones pueden variar de forma dinámica y en función del software:

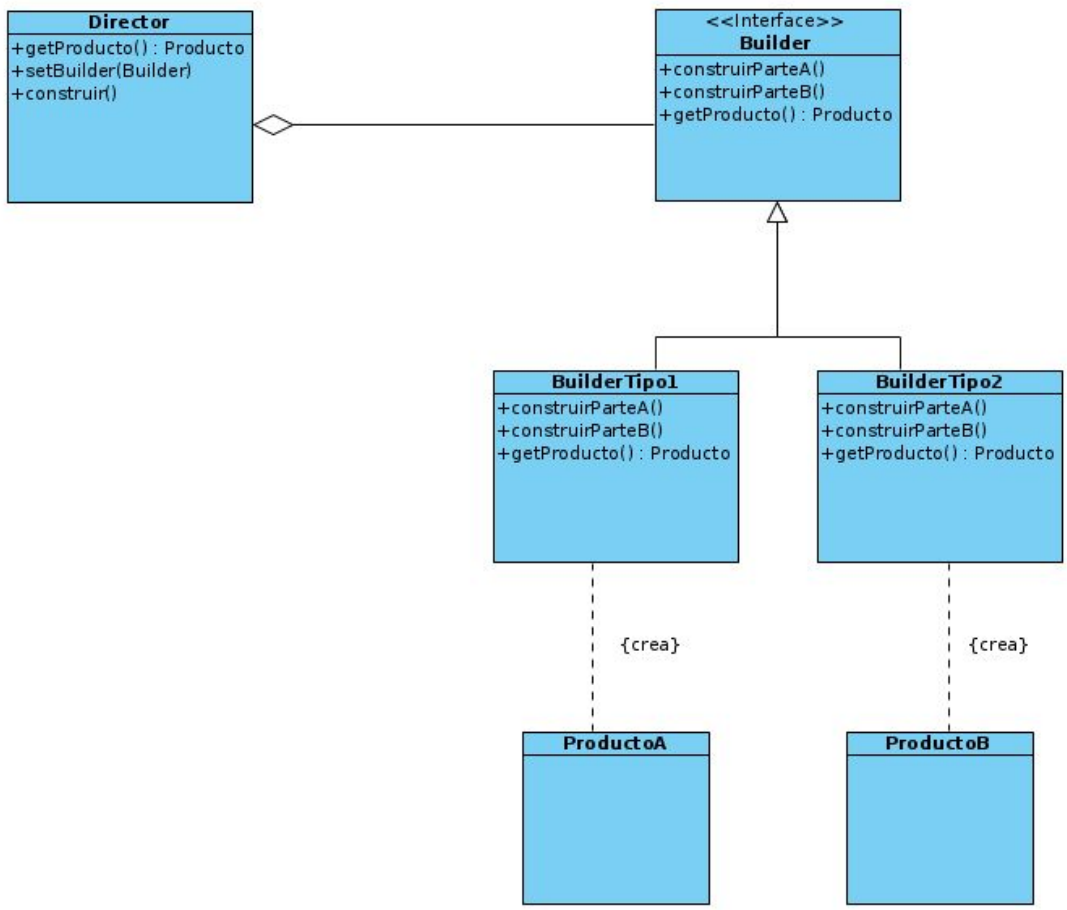

Modifica
MOP
MetaObjetoA
MetaObjetoB
Nivel Meta
Proporciona información
Nivel base
Usa
Usa
InterfazUsuario
ComponenteA
ComponenteB
Usa
Provee acceso a

Builder

Es un patrón creacional que sirve para construir objetos completos. Para ello se crean individualmente cada una de las partes que lo componen y se juntan, devolviendo así la instancia que la aplicación requería.

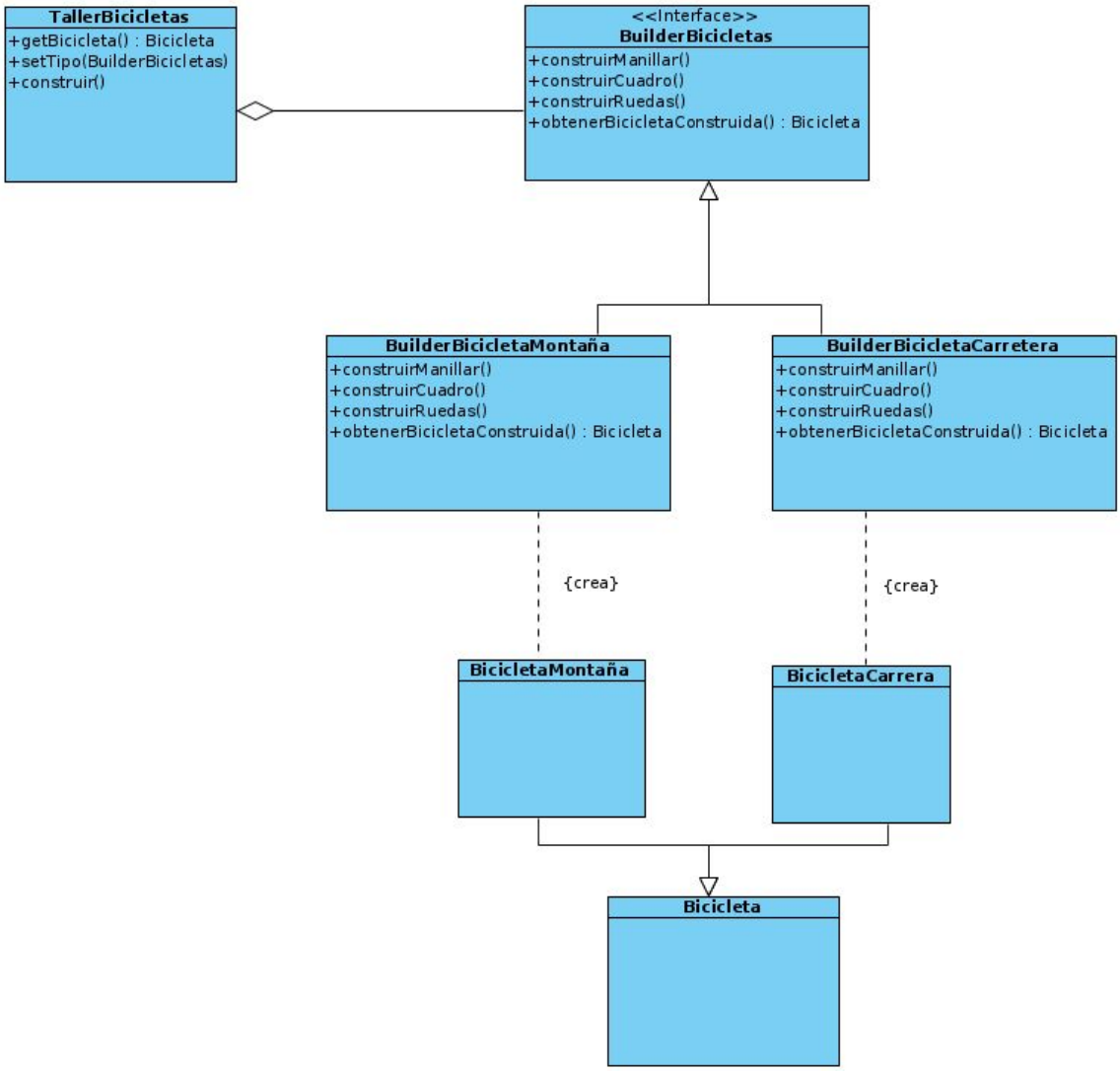
Cómo utilizarlo:

- Se crea la interfaz builder, la cual tiene métodos para crear cada una de las partes necesarias de la instancia.
- Se implementa tantas veces la interfaz anterior como tipos de objetos necesitemos crear.
- A la clase que utiliza Builder para crear las partes de un objeto se le denomina Director. La clase Director establece el tipo concreto de Builder a utilizar, y le pide que fabrique cada una de las partes necesarias, y permite obtener la instancia creada en el builder.



Ejemplo:

- Necesitamos crear las bicicletas que componen carreras. Estas pueden ser de montaña o de carrera.
- Creamos la interfaz builder, que llamaremos BuilderBicicletas. Tendrá los siguientes métodos sin iimplementar:
    - crearRuedas
    - crearCuadro
    - crearManillar
- Implementamos BuilderBicicletas centrándonos en los dos tipos de carreras que queremos que se permitan realizar. Para ello creamos las extensiones BuilderBicicletasMontaña y BuilderBicicletasCarrera. En ellos redefinimos los métodos para que creen las piezas necesarias para los dos tipos de bicicletas.
- Creamos el director, que en este caso llamaremos TallerBicicletas. TallerBicicletas contará con un método para asignar el tipo de BuilderBicicletas, otro para obtener la bicicleta creada del builder, y otro para crearla. En el método para crearla tendríamos que llamar a los métodos crearRuedas, crearCuadro y crearManillar del builder.
- En la aplicación cliente podríamos solicitar al TallerBicicletas el tipo deseado, que cree la bicicleta y que nos la devuelva.


TallerBicicletas
+getBicicleta() : Bicicleta
+setTipo(BuilderBicicletas)
+construir()
<<Interface>>
BuilderBicicletas
+construirManillar()
+construirCuadro()
+construirRuedas()
+obtenerBicicletaConstruida() : Bicicleta
BuilderBicicletaMontaña
+construirManillar()
+construirCuadro()
+construirRuedas()
+obtenerBicicletaConstruida() : Bicicleta
BuilderBicicletaCarretera
+construirManillar()
+construirCuadro()
+construirRuedas()
+obtenerBicicletaConstruida() : Bicicleta
{crea}
{crea}
BicicletaMontaña
BicicletaCarrera
Bicicleta