TDD Resumen

La Programación **Imperativa** describe instrucciones a ejecutarse paso a paso para variar el estado del programa. El estado final debería de ser la solución al problema. Describe de forma explícita qué pasos deben llevarse a cabo y en qué secuencia para llegar a la solución final.

La Programación **Declarativa** describe el problema que se quiere solucionar. El sistema usa la descripción para hallar un algoritmo que resuelva el problema.

La programación **imperativa** se centra en el **“cómo**”, y la **declarativa**, en el **“qué**”.

¿Que hay que hacer si todo lo que tenemos es imperativo?

* Alterar las entradas implícitas
* Invocar la subrutina
* Revertir los efectos colaterales de la subrutina
* Revertir los cambios en el primer paso.

**Programación funcional + Objetos**:

El objeto es el estado unido a un comportamiento.

La única incompatibilidad está en que en vez de mutar objetos creamos objetos nuevos.

Tenemos multimétodos, donde es polimórfico en todos los argumentos a la vez.

Functional core, imperative shell

Limita las capacidades imperativas a las partes que necesitan tenerlas

El functional core tiene la lógica de dominio, no tiene acceso al mundo exterior, y es fácil de

testear.

La shell imperativa tiene la lógica de entrada/salida y es difícil de testear.

**Redux**

Los reduce generalmente se aplican a una secuencia espacial. Se puede aplicar también a una secuencia temporal.

Los efectos clave son que los **eventos son inmutables**, el **estado es inmutable**, y la **función de avance de estado es predecible**. Todo puede volver a usarse sin distinción si es la primera o enésima vez. (Transiciones reproducibles y time travel debugging)

Es una forma en que se comuniquen las diferentes vistas con un core funcional.

Es en cierta forma en lo que se basa React.

Se tienen reducers que van cambiando la forma del estado.

**Patrones de diseño**

**Patrones creacionales**

Las abstractas arriba, las dependientes - inestables - abajo. Esto muestra la inversión en la cadena de dependencia.

Se usan cuando hay que aislar la creación de instancias.

Resuelven el encapsulamiento, ocultan y ordenan la creación de instancias, vuelven al sistema independiente del proceso de creación.

No publicar cuestiones que no le interesan a otro paquete.

Se vuelven más importantes a medida que se pasa de usar herencia a composición de objetos.

Nos dan mucha flexibilidad en cuanto a que se crea, quien lo crea, como es creado, y cuando se crea.

**Factory Method**

*Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.*

* La forma más simple de ocultar la creación de un objeto es usando un método que devuelve

un objeto.

* Si aparecen nuevas clases del tipo1, extendemos la de la fábrica original, de manera tal de

ocultar la creación de cada uno de los tipos en métodos de las clases diferentes.

* Presenta la dificultad de mantener 2 jerarquías de clases en paralelo.
* Se bautiza como factory method a cualquier clase que tenga un método para crear otro

objeto y devolverlo.

* Su intención es crear una interfaz para la creación de un objeto que permita a las subclases

decidir qué clase instanciar.

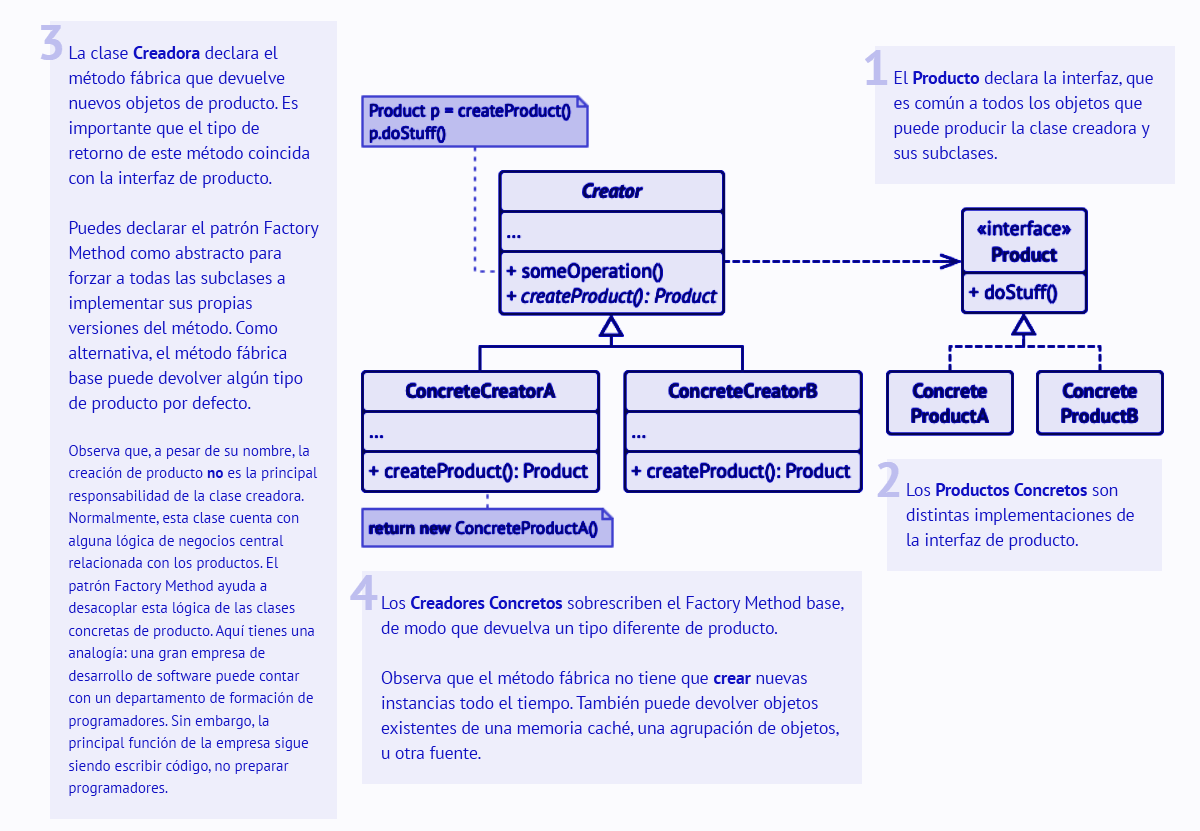
* Elimina la necesidad de atar clases específicas de la aplicación al código.
* Crear en una manera descriptiva las instancias que no son descriptivas. (Clase con muchos

constructores, se complica elegir cual). Necesitamos sacar la ambigüedad de esto. Una forma

de arreglar esto es proveer métodos de clase que tengan nombres descriptivos y constructores

protegidos. Cada uno de los métodos sería un factory method. (no es la version propuesta

por Gamma, es válida igual. )) - Ejemplo



**Abstract Factory**

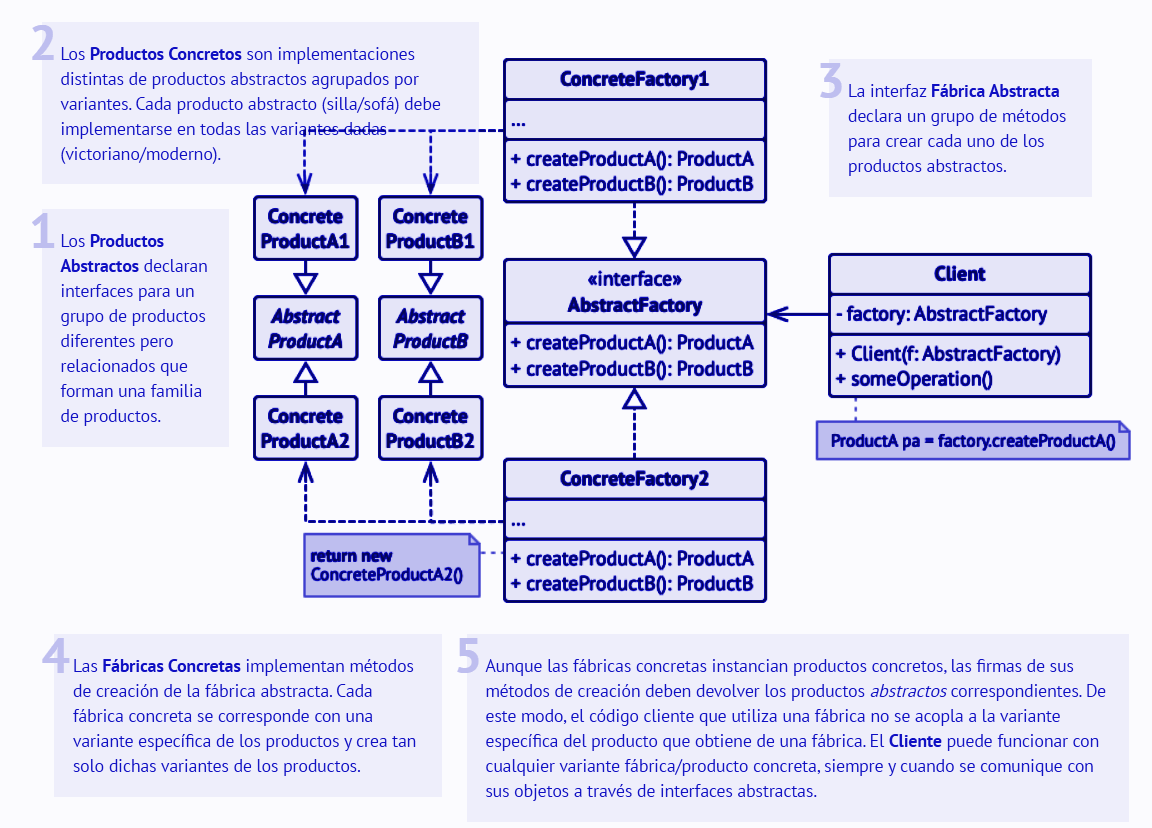
*Permite producir familias de objetos relacionados a partir de una interfaz sin especificar sus clases concretas.*

* Logramos un esquema en el cual se crean objetos concretos sin violar la inversión de la

dependencia.

* Busca proveer una interfaz para crear familias de objetos relacionados o dependientes entre sí sin especificar la clase en concreto.
* Se usa cuando un sistema debe de ser independiente de cómo se crean los objetos, cuando un sistema debe de configurarse con una familia de muchos objetos de muchas.
* Aísla las clases concretas.
* Los clientes interactúan sólo con la interfaz.
* Vuelve fácil el cambio de productos de familias.
* Promueve la consistencia entre los productos.
* Se vuelve difícil soportar nuevos tipos de productos. Esto es porque la interfaz de la abstract

factory fija el conjunto de productos.

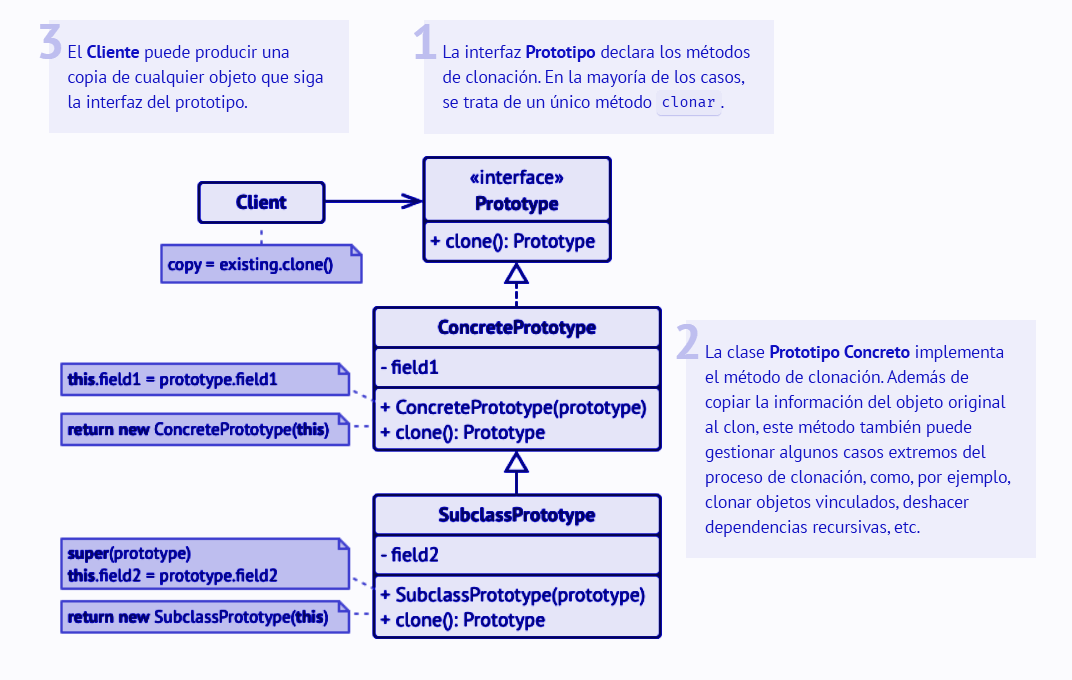


## 

## **Prototype**

*Permite copiar objetos existentes sin que el código dependa de sus clases.*

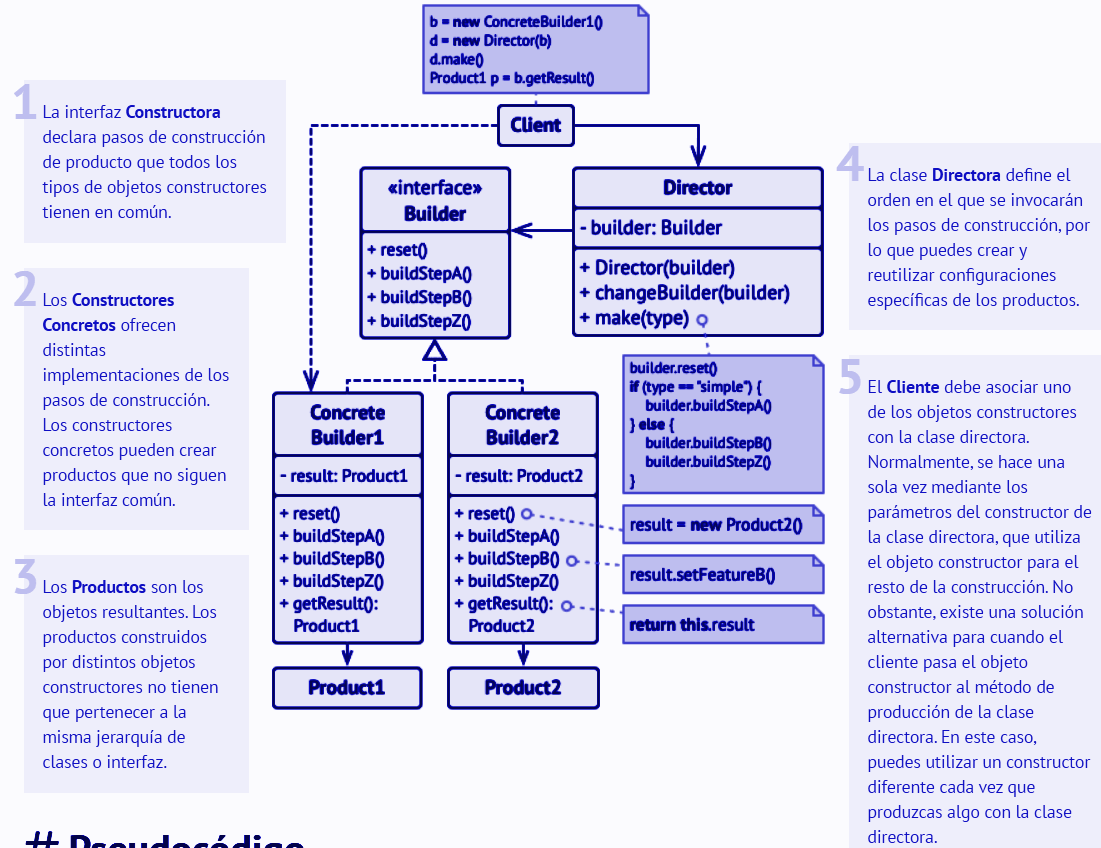
* En vez de crear clona objetos. Nos evita tener que mantener la jerarquía paralela de la factory.
* La intención es especificar los tipos de objetos a crear a partir de un prototipo, y crear nuevos objetos a partir de copiar el prototipo.
* El patrón declara una interfaz comun para todos los objetos que soportan la clonacion permitiendo clonar un objeto sin acoplar el código a la clase de ese objeto.
* Se usa cuando un sistema debe de ser independiente de cómo sus productos son creados, las clases de productos a instanciar se especifican en tiempo de ejecución, se quiere evitar jerarquías de fábricas paralelas a la de productos, las instancias de una clase sólo pueden tener pocas combinaciones de estado… Esto esconde las clases concretas del cliente.
* Se pueden agregar y quitar prototipos en tiempo de ejecución.
* Reduce las subclases



**Builder**

*Permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.*

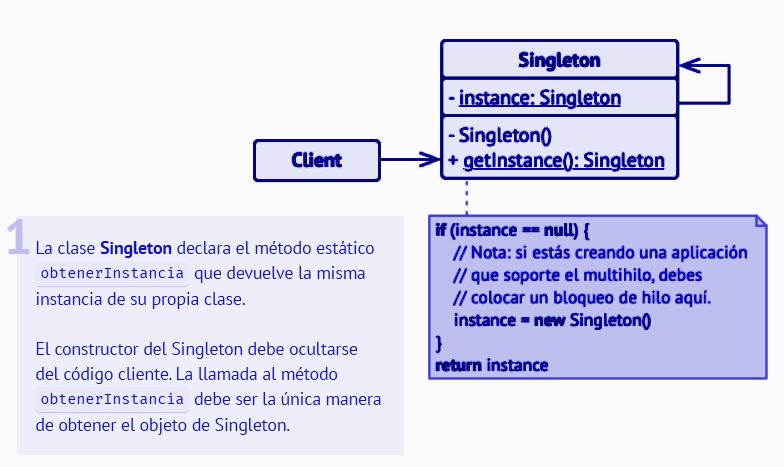
* Variante del abstract factory. Cuando invoco a crear un objeto, invoca a la creación de cada parte del objeto complejo.
* Busca separar la construcción de un objeto complejo de la representación, de forma tal de que el mismo proceso crea diferentes representaciones.
* Sugiere que saques el código de construccion del objeto de su propia clase y lo coloques dentro de objetos independientes llamados **constructores**
* Se usa cuando el algoritmo de creación deba de ser independiente del objeto y la construcción debe de permitir diferentes representaciones.
* Esto permite cambiar la representación interna de un producto y esconde la representación interna del producto con la interfaz.
* Aísla el código para construcción y representación. Esto mejora la modularidad al encapsular el proceso.
* Nos da un control más fino sobre el proceso de creación. (Lo hace al objeto paso a paso) y no se requiere que se incoquen a todos los pasos, solo aquellos que necesites para configurar/crear a tu objeto particular



**Singleton**

*Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.*

* Se usa cuando se quiere que exista una única instancia de una clase y proveer un único punto de acceso a ella.
* Permite un punto de acceso controlado.
* Es una mejora respecto de variables globales. (En cierta forma lo es todavía)
* Permite variar el número de instancias.
* Es más flexible que los métodos de clase.
* Se puede extender la clase.
* La clase es responsable de crear la unica instancia
* Los usos más frecuentes son en caches, configuraciones y pulls.
* Patrón fácil pero intrusivo (ensucia la clase)



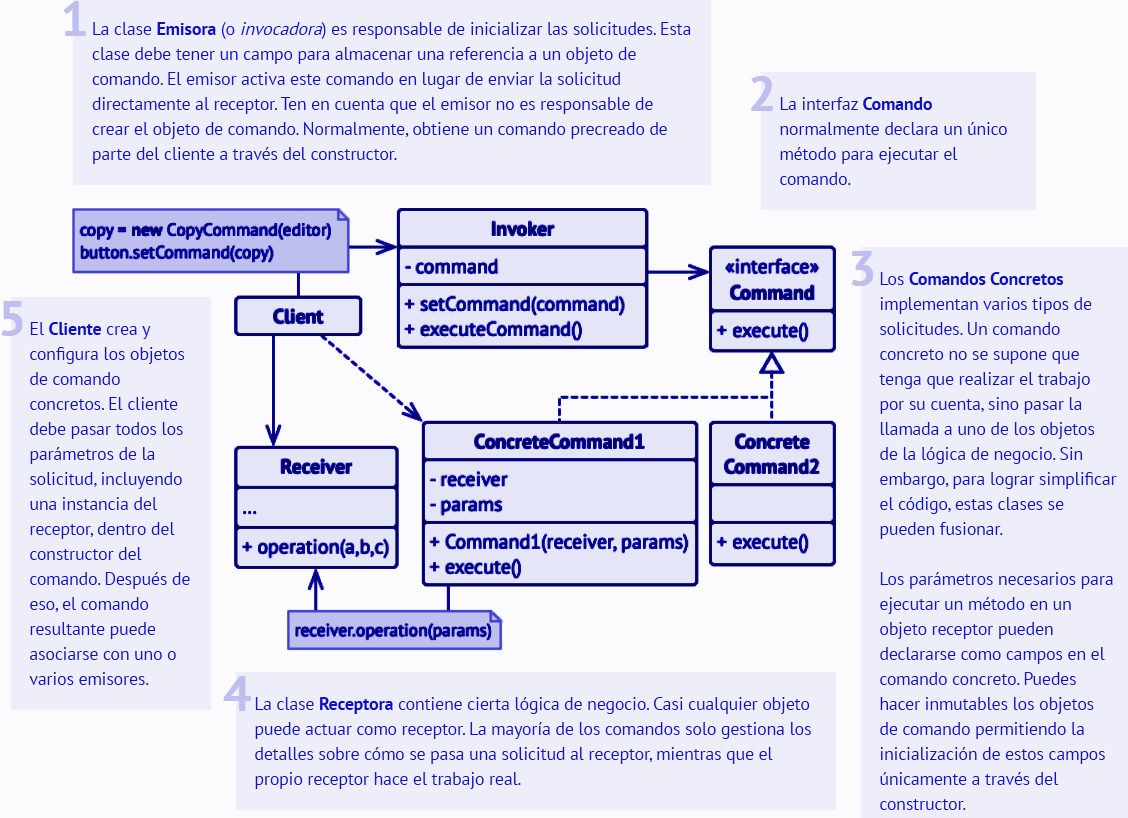
**Patrones de comportamiento:**

Estos patrones tratan con algoritmos y la asignación de responsabilidades entre objetos.

**Command:**

*Command es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud.*

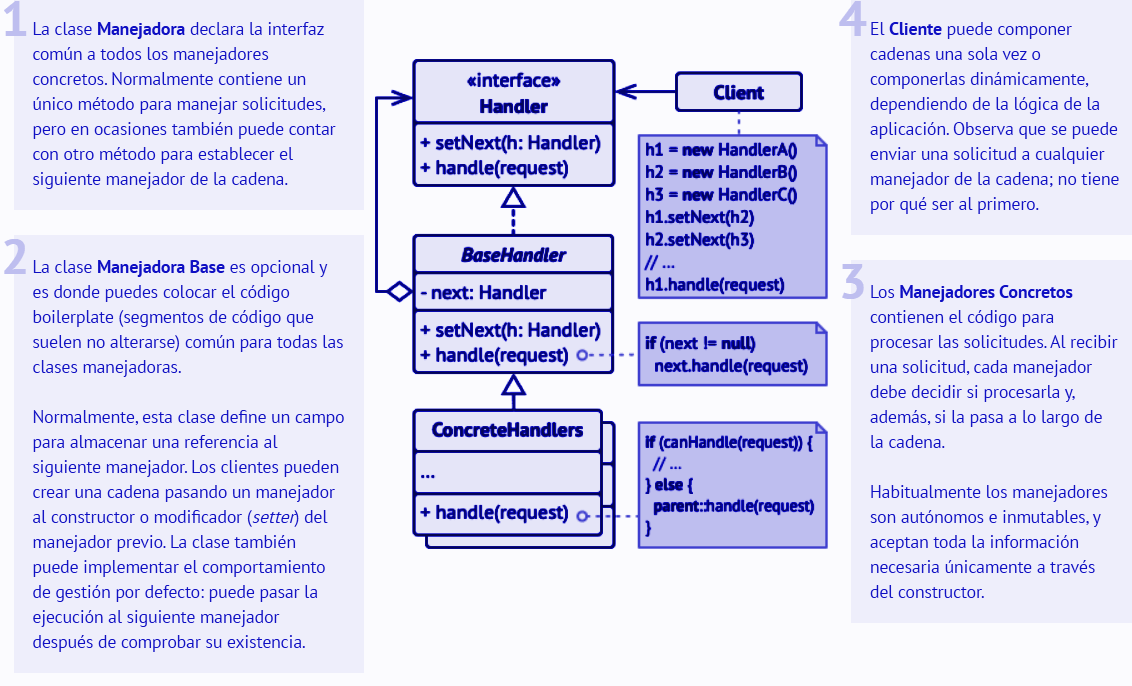
* La intención es encapsular la acción de un objeto.
* Se usa cuando se quiere parametrizar objetos mediante una acción a ejecutar, especificar, encolar, y ejecutar acciones en diferentes momentos, soportando deshacer la operación.
* Desacopla el objeto que solicita un servicio de aquel que lo presta. Versión en objetos de una función Callback
* Se pueden componer comandos con un composite.
* Se vuelve fácil agregar nuevos comandos, no se tiene que cambiar clases existentes



**Chain of Responsibility**

*Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.*

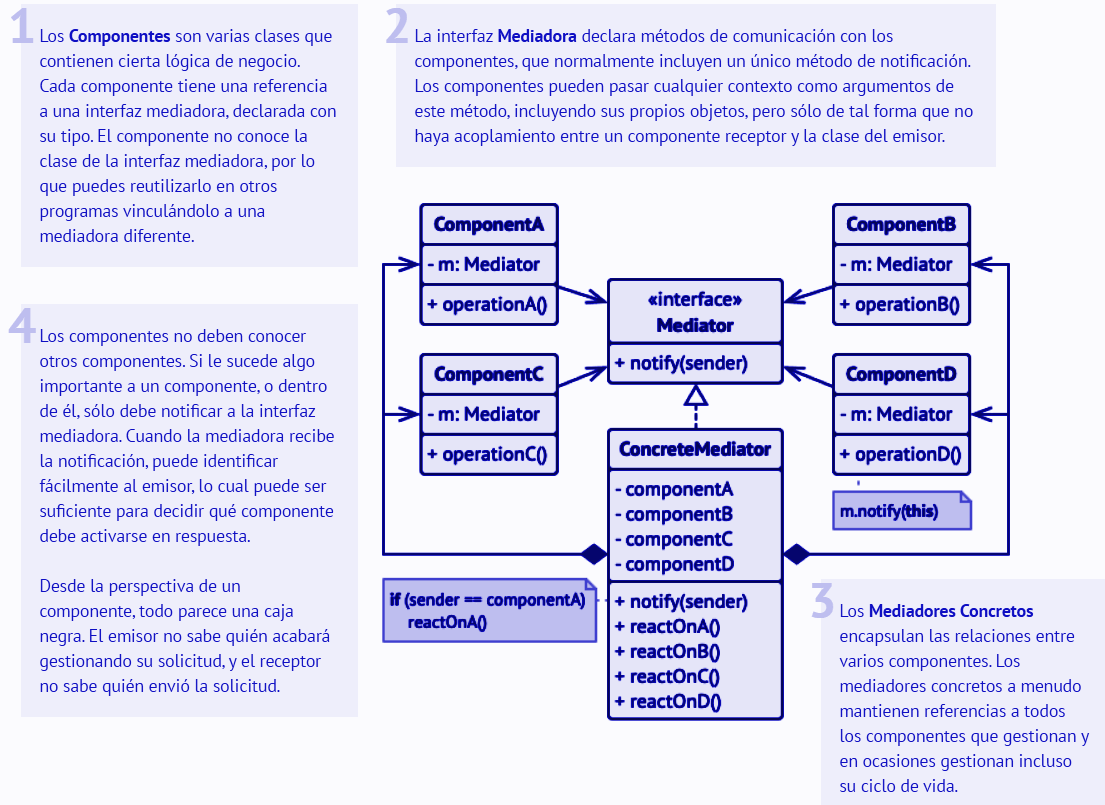
* La intención es desacoplar quien envía un mensaje de quien lo ejecuta. Los objetos lo van pasando.
* Se usa cuando más de un objeto puede manejar un mensaje y este objeto no se conoce a priori, se quiere mandar un mensaje a varios objetos sin especificar cuales en específico, el conjunto de objetos que manejan el mensaje se especifica dinámicamente.
* Va pasando la posta hasta que lo trate alguien. Desacoplar quien solicita un servicio de un conjunto.
* Se puede ver también como una función de orden superior que toma una función, y si no puede resolver lo delega a la función.
* Reduce el acoplamiento. Libera a los objetos al saber quién lo va a manejar al mensaje.
* Da flexibilidad en asignar responsabilidades a los objetos.
* El receptor no está garantizado, por lo que puede no ejecutarse si no se configura correctamente.
* Este patrón se basa en transformar comportamientos particulares en objetos autonomos llamados *manejadores*
* Estos manejadores están vinculados por medio de una cadena, y cada uno tiene la referencia al siguiente. La solicitud va pasando por todos los manejadores y un manejador puede decir cortar el flujo (no pasar más la solicitud) o seguir pasandola



**Mediador**

*Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador*

* La intención es definir un objeto que encapsule como un conjunto de objetos interactúan entre sí.
* Se usa cuando un conjunto de objetos se comunican de forma bien definida pero compleja.
* Busca evitar una cantidad de dependencias enormes entre todos los componentes. Este mediador elimina las dependencias cruzadas, pero tiene la desventaja de que todos conocen al mediador, y que el mediador conoce a todos (Tenemos una dependencia circular). Es fácil agregar componentes adicionales porque la lógica está concentrada en el mediador.



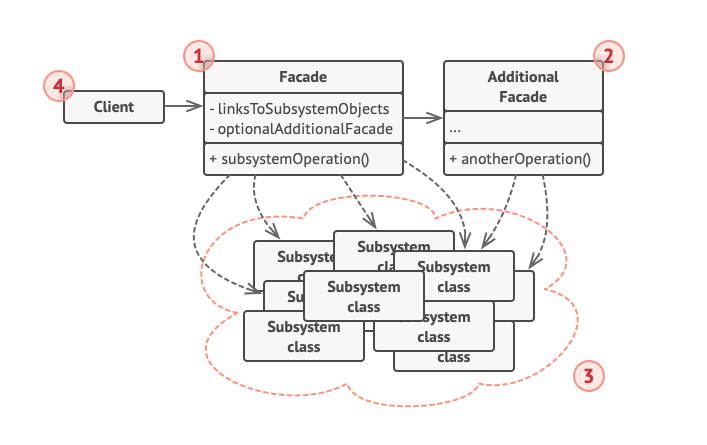
**Patrones de control de acceso**

* Se encargan sobre cómo las clases y objetos se componen para formar estructuras mas grandes. (Structural Patterns)
* Compone objetos para lograr nuevas funcionalidades. (Structural Patterns)

**Facade**

*Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.*

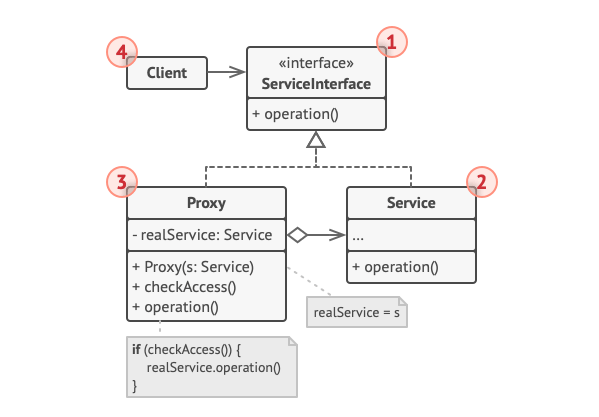
* La intención es proveer una interfaz unificada a un conjunto de interfaces de un subsistema.
* Se usa cuando queremos una interfaz simple a un subsistema complejo, hay muchas dependencias entre clientes y la implementación de las clases de una abstracción, queremos una capa entre los subsistemas (entry point).
* Escribimos código para acceder servicios externos y no contaminar el contexto de nuestro sistema. Aparece la segregación de interfaces.
* Tener un acceso de más alto nivel.
* Reducir el acoplamiento.
* Protege a los clientes de las componentes de los subsistemas.
* Promueve weak coupling entre los subsistemas y los clientes.
* No previene a las aplicaciones el uso de los subsistemas si quieren. Permite elegir entre la facilidad de uso y generalidad.



**Proxy**

*Proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.*

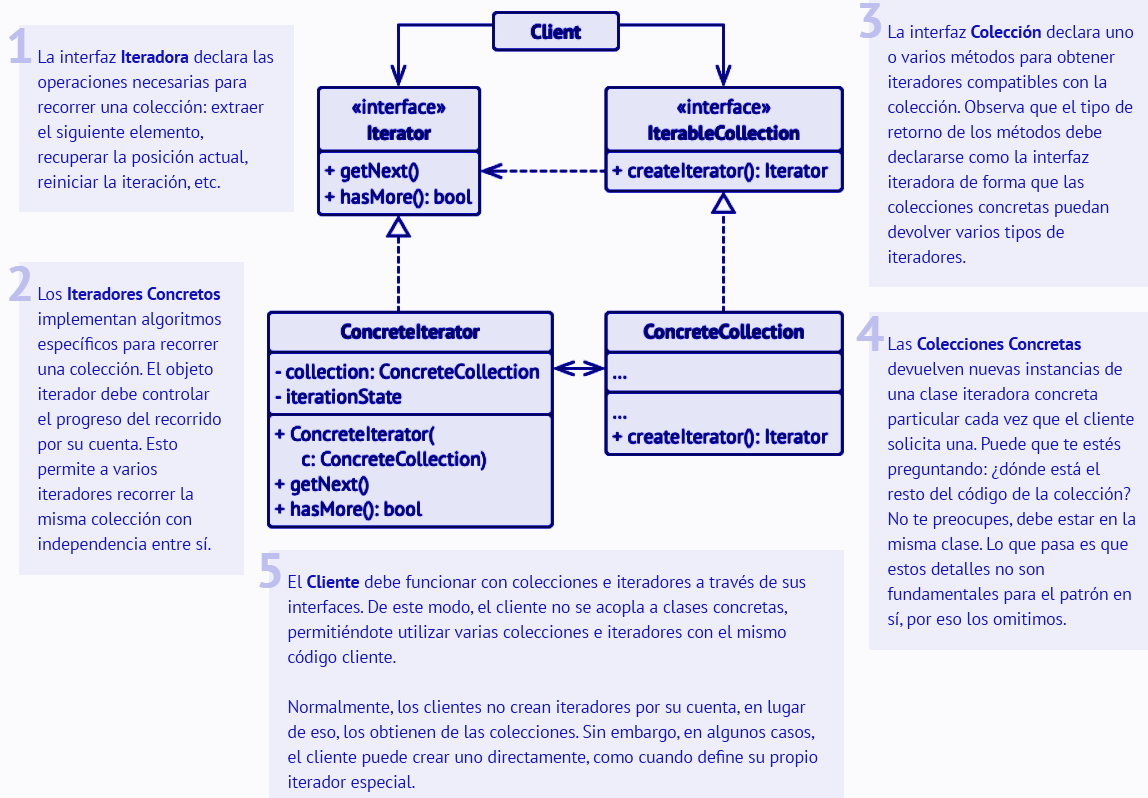
* Su intención es proveer un sustituto a otro objeto para controlar el acceso.
* En los métodos de proxy se decide que puede hacer la clase que llama respecto de la clase oculta.
* Controla acceso a determinados tipos de recursos, esto posterga el costo de la creación e inicialización hasta que se necesite.
* Proxy remoto para objetos separados de lugar.
* Proxy virtual: Se usa en las arquitecturas de tipo EA. Lazy load, creo en memoria un proxy, el cliente se cree que tiene un objeto completo, pero en realidad voy trayendo lo que necesito a medida que se pide.
* El proxy dio lugar al garbage collector, de alguna manera controla la creación y destrucción de objetos.



**Iterator (Behavioral Patterns)**

*Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).*

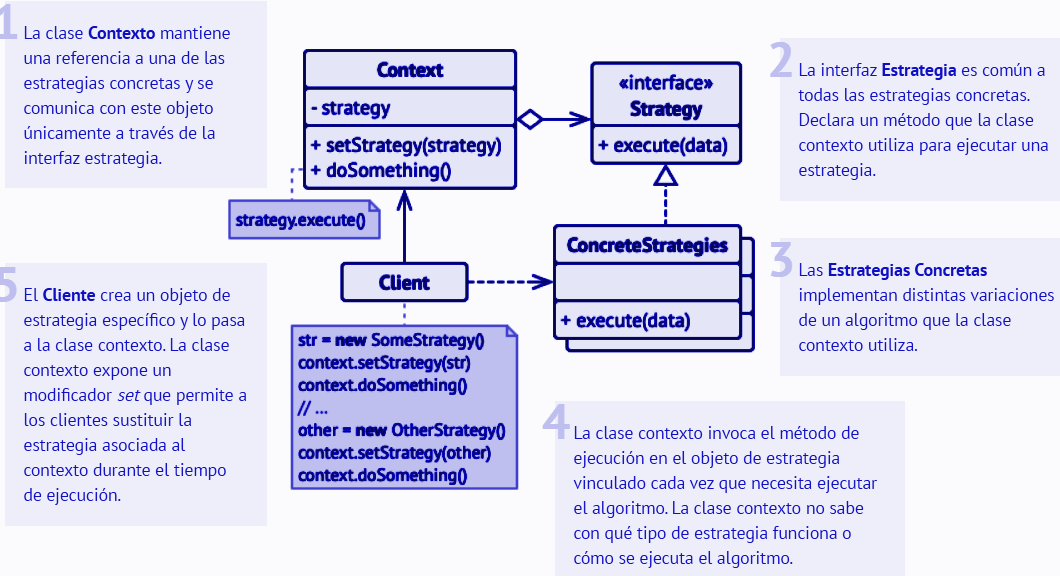
* La motivación es proveer una forma de acceder a los elementos de un aggregate object se forma secuencial sin exponer la representación.
* Permite múltiples formas de recorrer el objeto.
* Provee una interfaz uniforme para el recorrido. (polymorphic iteration)
* Sirve para no exponer los datos de una colección directamente, los accedo a través de un iterador.



**Strategy (Behavioral Patterns)**

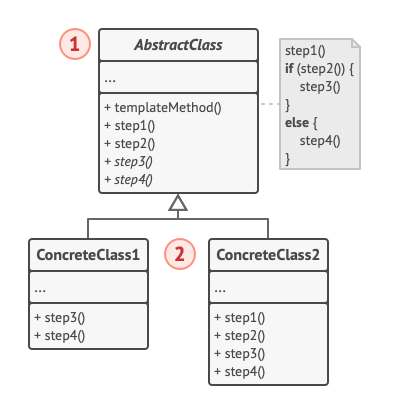
*Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables*.

* Se usa cuando tengo muchas formas de hacer una cosa. Se dice que encapsula un algoritmo y permite variarlo independientemente de los clientes.
* Cambia el algoritmo completo.
* Recibe el contexto que es la clase que se va a usar. Contiene la información necesaria para resolver la estrategia.
* Se aprovecha el tiempo de ejecución.
* De acuerdo a la estrategia que use estoy variando lo que hago.



**Template Method**

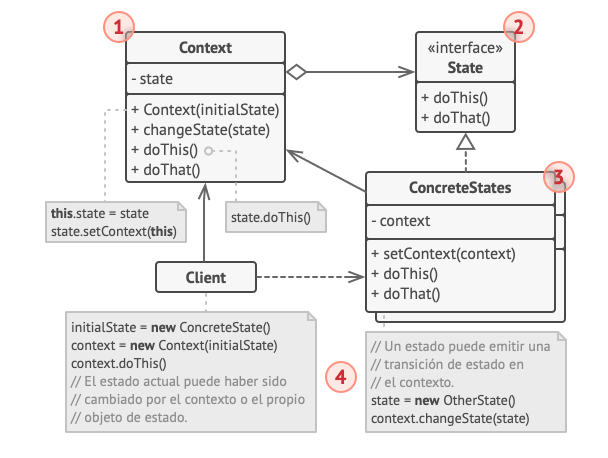
*Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.*

* El algoritmo está prefijado, es decir, define el esqueleto de un algoritmo
* Cada una de las clases redefinen el paso correspondiente
* El objeto instanciado ejecuta el algoritmo
* Muy usado en frameworks
* Se trata de tener diversidad y resolverlo en tiempo de compilacion
* Fundamental para reusar código
* Este patrón sugiere que dividas un algortimo en una serie de pasos, conviertas esos pasos en metodos y coloques una serie de llamadas a esos metodos dentro de un unico metodo planilla. 

**State(Behavioral Patterns)**

*Permite a un objeto alterar su comportamiento cuando su estado interno cambia. parece como si el objeto cambiará su clase.*

* Está relacioado con que los objetos se conporten de acuerdo al estado en que se encuentran
* El estado de un objeto depende de su historia
* Se suele utilizar cuando aparecen muchas condiciones
* Un beneficio es que localiza el comportamiento dependiendo del estado y divide dicho comportamiento en diferentes estados.
* El objeto original almacena una referencia al estado actual y delega todo el trabajo relacionado con el estado a ese objeto

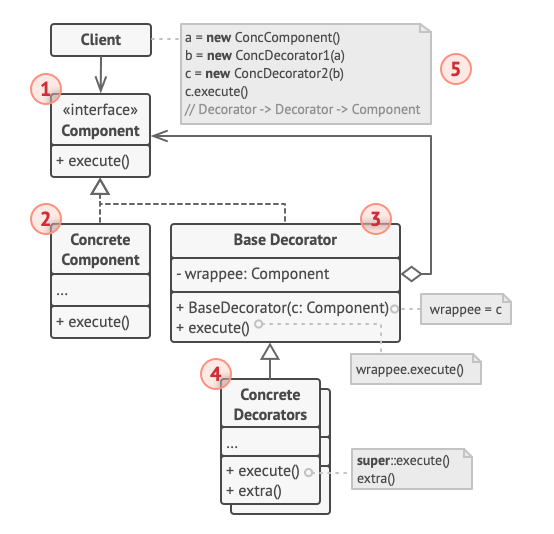


**Extension de servicio**

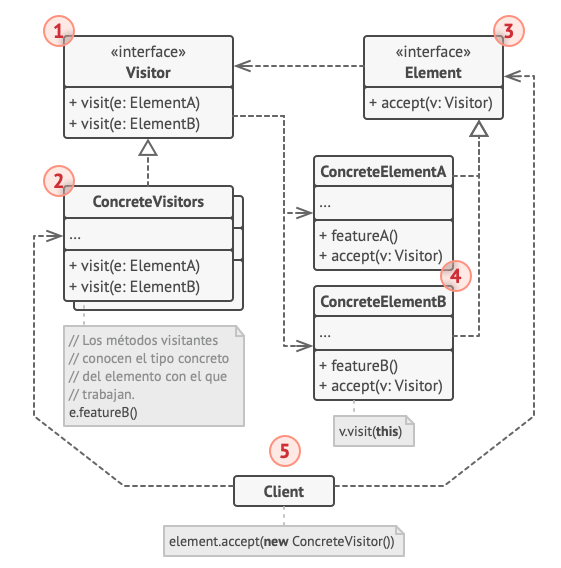
**Decorator**

*Te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.*

* Está orientado a objetos “livianos”, encapsulan más comportamiento que atributos
* Extiende el servicio en tiempo de ejecucion, le agrega o quita responsabilidades
* Puede tener clases hijas que redefinen su comportamiento
* Es más flexible que la herencia
* Da mucha reusabilidad
* Generalmente se usa con composite

**Visitor**

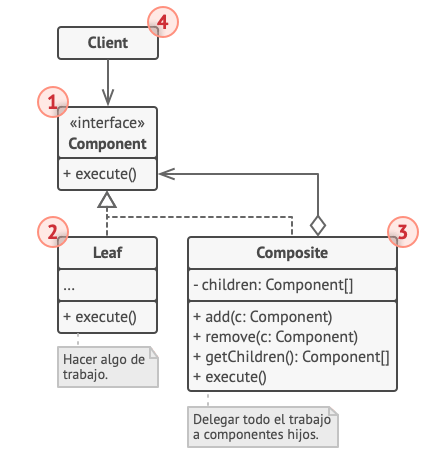
*Permite separar algortimos de los objetos sobre los que operan*

* Lo usamos para agergar metodos/funciones a una clase
* Se decie que es un doble dispatch
* Se puede usar cuando un objeto contiene muchas clases de objetos con diferentes interfaces y queremos realizar acciones que dependen de las clases concretas.
* Este patrón sugiere que coloques un nuevo comportamiento/algortimo en una clase separada llamada visitante.

**Varios**:

**Composite:**

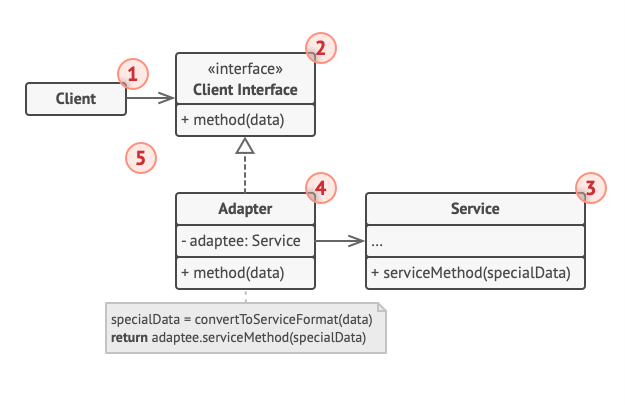
*Te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.*

* Ejemplos de uso se ven en procesamiento de texto y filtros
* Compone objetos en estructuras -similares a árboles - para representar jerarquias.
* Simplifica al cliente ya que se vuelve indistinguble al grupo de objetos de un solo objeto 

**Adapter**

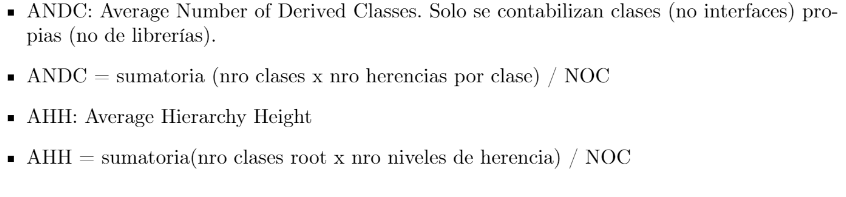
*Permite la colaboración entre objetos con interfaces incompatibles.*

* Está orientado a reutilizar código
* Cuando es necesario adaptar una clase para que funcione en el conexto que necesito
* Suele usarse para adaptar los metodos de algun sistema externo que estamos usando para que un facada acceda a ese sistema.
* Convierte una interfaz en otra que espera el cliente

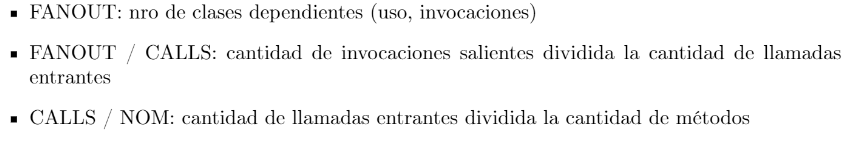


**Metricas:**

**Profundidad de herencia** : Indica el número de clases diferentes que heredan entre sí, hasta la clase base. La profundidad de herencia es similar al acoplamiento de clases en que un cambio en una clase base puede afectar a cualquiera de sus clases heredadas. Cuanto mayor sea este número, más profunda será la herencia y mayor será la posibilidad de modificaciones de clase base para dar lugar a un cambio importante. En Profundidad de herencia, un valor bajo es bueno y un valor alto es incorrecto.

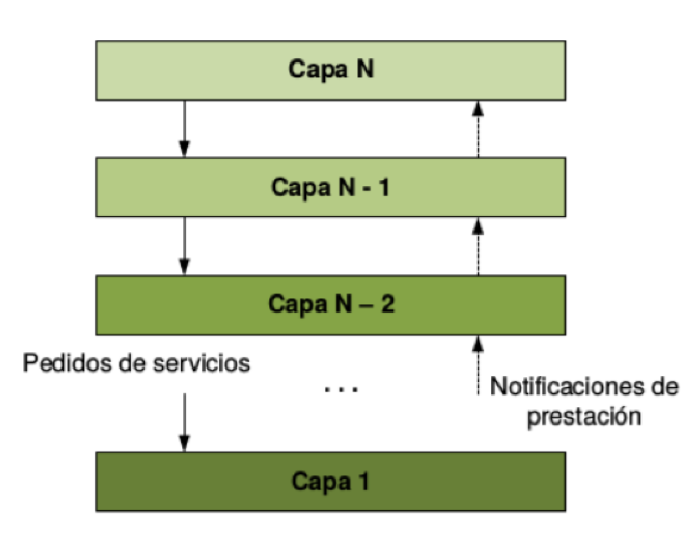


**Acoplamiento de clases**: mide el acoplamiento a clases únicas a través de parámetros, variables locales, tipos de valor devuelto, llamadas de método, instancias genéricas o de plantilla, clases base, implementaciones de interfaz, campos definidos en tipos externos y decoración de atributos. El buen diseño de software dicta que los tipos y métodos deben tener una cohesión alta y un acoplamiento bajo. El acoplamiento alto indica un diseño difícil de reutilizar y mantener debido a sus numerosas interdependencias en otros tipos. Para obtener más información, vea Acoplamiento de clases.



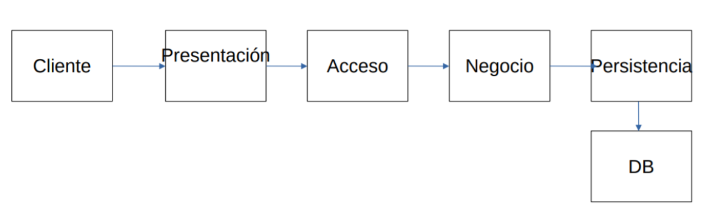
**Tamaño**: mide el tamaño de nuestro sistema que puede ser por medio de cantidad de líneas de código. Permite medir la cantidad de líneas por función. Si este número es muy alto, se debe revisar el código para ver si hay responsabilidades mixtas o es confuso el bloque de código.

**Patrones de Arquitectura**

Layer 

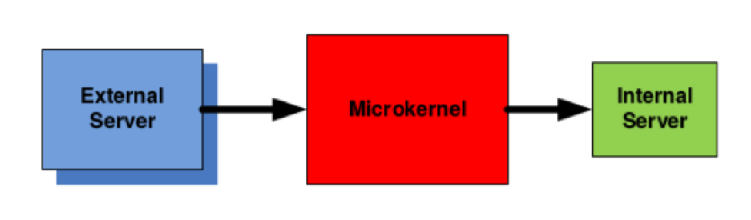
* Se utiliza cuando aparece conceptos de distintos niveles de abstraccion
* En este patrón se tiene diferentes capas con distintos niveles de abstraccion.
* Cada capa utiliza un servicio de la capa superior. Permite reutilizar partes.
* Ventajas: reuso, cambiabilidad, desarrollo simultáneo
* desteventajas: diseño elaborado

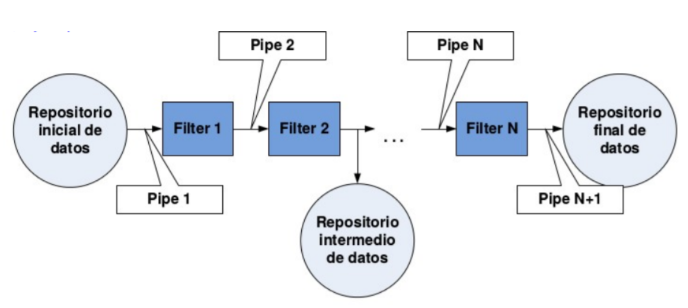
EA



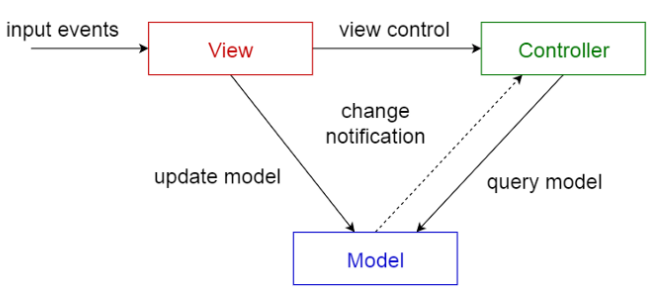
* Es un mega patrón compuesto por muchos patrones
* Es conveniente usarlo cuando hay un sistema cliente-servidor, usuarios concurrenres, multiples interfaces de usuario, reglas de negocio y datos persistenetes.
* Ventajas: reuso y cambiabilidad
* Desventaja: mayor trabajo

Microkernel

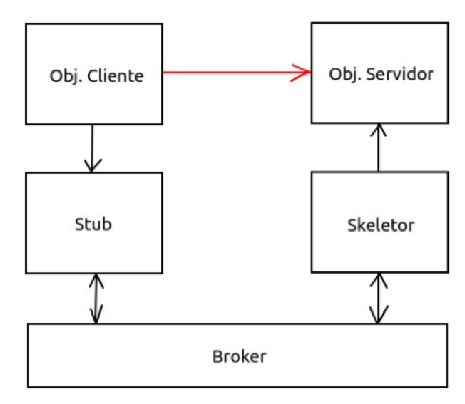
* Se utiliza cuando el sistema debe ser capaz de adaptarse a los cambios en los requisitos del sistema. 
* Es utilizado en sistemas de larga vida que deben evolucioanr ante cambios de distinto tipo con diferente frecuencia
* Esta compuesto por external servers, microkernel e internal server
* Ventajas: soporta cambios con menor impacto
* Desventaja: un diseño más elaborado

Pipe Filter

* Se utilixa cuando se tiene que procesar una serie de datos, y ese procesamiento se puede descomponer en N procesamientos atomicos.
* Es reutilizable
* Se distribuyen los procesamientos en filtros
* Ventaja: reuso y rapido prototipado
* Desventaja: no se puede usar en sistemas críticos

MVC

* Conviene usarlo cuando es necesario contar con interfaces de usuarios multiples y diferentes.
* El usuario interactua con el modelo a traves de los controladores
* Ventajas: reusabilidad y facilidad de cambios y extension
* Desventajas: tiene un mayor diseño

Broker

* Se utiliza para estructurar sistemas de software distribuidos con componentes desacoplados que interactúan por invocaciones de servicios remotos.
* El broker se encarga de proporcionar la conexion entre componentes
* Se utiliza cuando se tiene un procesamiento distribuido
* Ventajas: cambiabilidad, portabilidad y reuso
* Desventajas: baja eficiencia, dificultad de desarrollo y no tolera fallas