Esquema TDS

Jose Antonio Lorencio Abril

1 Patrones GRASP

Patrón EXPERTO

Las responsabilidades deben recaer sobre la clase que tiene la información necesaria para cumplimentarla.

Para lograrlo podemos distribuir responsabilidades de forma homogénea y no crear clases dios.

- **Beneficios:**
  - Se conserva encapsulación
  - Produce alta cohesión

Patrón CREADOR

Es importante determinar qué clase es la responsable de crear una nueva instancia de una cierta clase.

Una clase B debe tener la responsabilidad de crear instancias de otra clase A si:

- B es una agregación de objetos de A
- B contiene objetos de A
- B registra instancias de A
- B hace un uso específico de los objetos de A
- B proporciona los datos de inicialización necesarios para crear un objeto de A

Este patrón nos proporciona los siguientes

- **Beneficios:**
  - Bajo acoplamiento

Patrón CONTROLADOR

Determina quién se encarga de manejar eventos externos a uno o más objetos controlador, que pueden representar el sistema o una funcionalidad concreta.

Actúa de fachada entre la capa de presentación (GUI) y la capa de dominio

Incluye, al menos, un método por cada operación atendida.

- **Beneficios:**
  - Separación modelo-vista
  - Posibilidad de capturar información sobre el estado de una sesión

2 Patrones de diseño

2.1 Patrones de Creación

Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y reutilización del código existente.

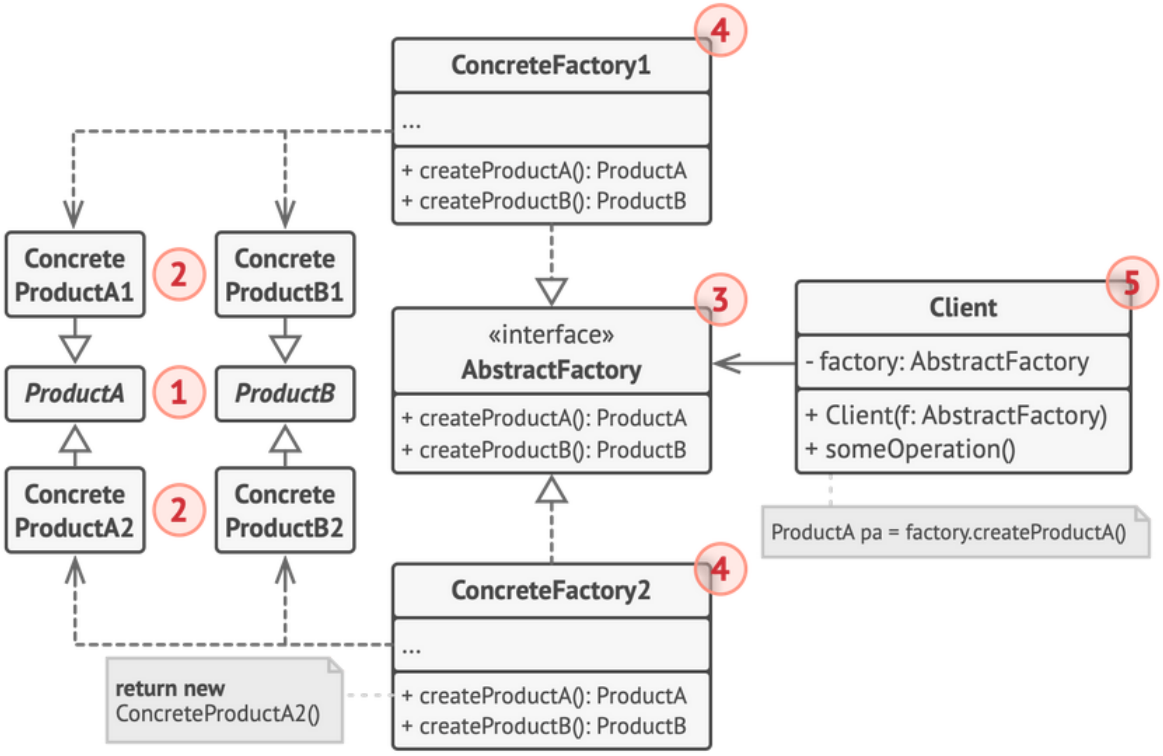
Patrón ABSTRACT FACTORY (Factoría Abstracta)

- Tiene como **propósito** proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar la clase concreta.
- Es **aplicable** cuando un sistema debería ser configurado para una familia familia de productos y ser independientes de familias y productos concretos, ya sea por no conocer las clases concretas de antemano o por querer permitir una futura extensibilidad
- **Consecuencias:**
  - Desacopla a la aplicación de las clases concretas de implementación
  - Facilita el intercambio de familias de productos
  - Favorece la consistencia entre productos
  - Un nuevo producto requiere modificar todas las factorías
  - **Principio de responsabilidad única:** podemos mover el código de creación de productos a un solo lugar, mejoran la mantenibilidad del código
- Se **implementan** como singleton. La factoría abstracta será una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos. Estos productos habrán sido implementados mediante una interfaz, y cada línea del producto implementará la concreción de la interfaz. La factoría abstracta devolverá productos abstractos representados por estas interfaces.

  Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz que hemos descrito. Así, una fábrica es una clase que devuelve productos de un tipo particular.

  El código cliente tiene que funcionar con fábricas y productos a través de las interfaces, de esta forma podemos cambiar el tipo de fábrica que le pasamos, así como la variante del producto, sin descomponer el propio código cliente.

  Ahora bien, si el cliente solo está expuesto a las interfaces abstractas, ¿cómo se crean los objetos de fábrica? Normalmente la aplicación crea un objeto de fábrica concreto en la etapa de inicialización. Justo antes, la aplicación debe seleccionar el tipo de fábrica.



Los **productos abstractos** declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.

Los **productos concretos** son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto debe implementarse en todas las variantes.

La interfaz **fábrica abstracta** declara un grupo de métodos para crear cada uno de los productos abstractos

Las **fábricas concretas** implementan los métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante de los productos. Crea productos de esa variante.

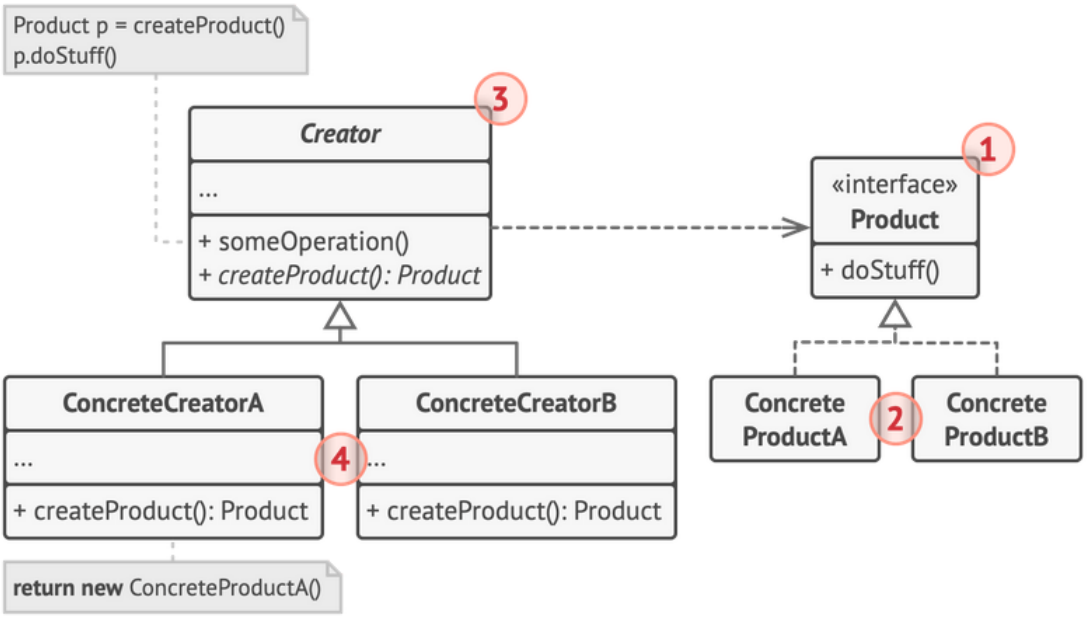
Aunque crean productos concretos, devuelven los productos abstractos correspondientes. Así, el código cliente no se acopla a la variante específica del producto que obtiene de una fábrica.

Patrón FACTORY METHOD (Método Factoría)

- **Propósito:** define un interfaz para crear un objeto, pero permite a las subclases decidir la clase a instanciar: instanciación diferida a las subclases.
- Viene **motivado** por las situaciones en las que una clase B que utiliza una clase abstracta A necesita crear instancias de subclases de A que no conoce.
- Se puede **aplicar** cuando una clase no puede anticipar la clase de objetos que debe crear.
  - Queremos ofrecer a los usuarios de nuestra biblioteca una forma de extender sus componentes internos
  - Queremos ahorrar recursos mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez
- **Consecuencias:**
  - Evita ligar un código a clases específicas de la aplicación
  - Puede suceder que las subclases de Creador solo se creen con el fin de la creación de objetos (Esto debe evitarse)
  - El método factoría puede ser invocado por un cliente, no solo por la clases Creador (esto se denomina **jerarquías paralelas**)
- Para la **implementación**, tenemos dos posibilidades:
  - Crear el método factoría abstracto
  - Crearlo con una implementación por defecto

  Se debe evitar crear subclases de creador con metaclases y cuando el método factoría puede tener un parámetro que identifica a la clase del objeto a crear.

  En java está la alternativa de utilizar la interfaz funcional Supplier<>.
- **Estructura**



El **producto** declara la interfaz, que es común a todos los objetos que pueden producir la clase creadora y sus subclases

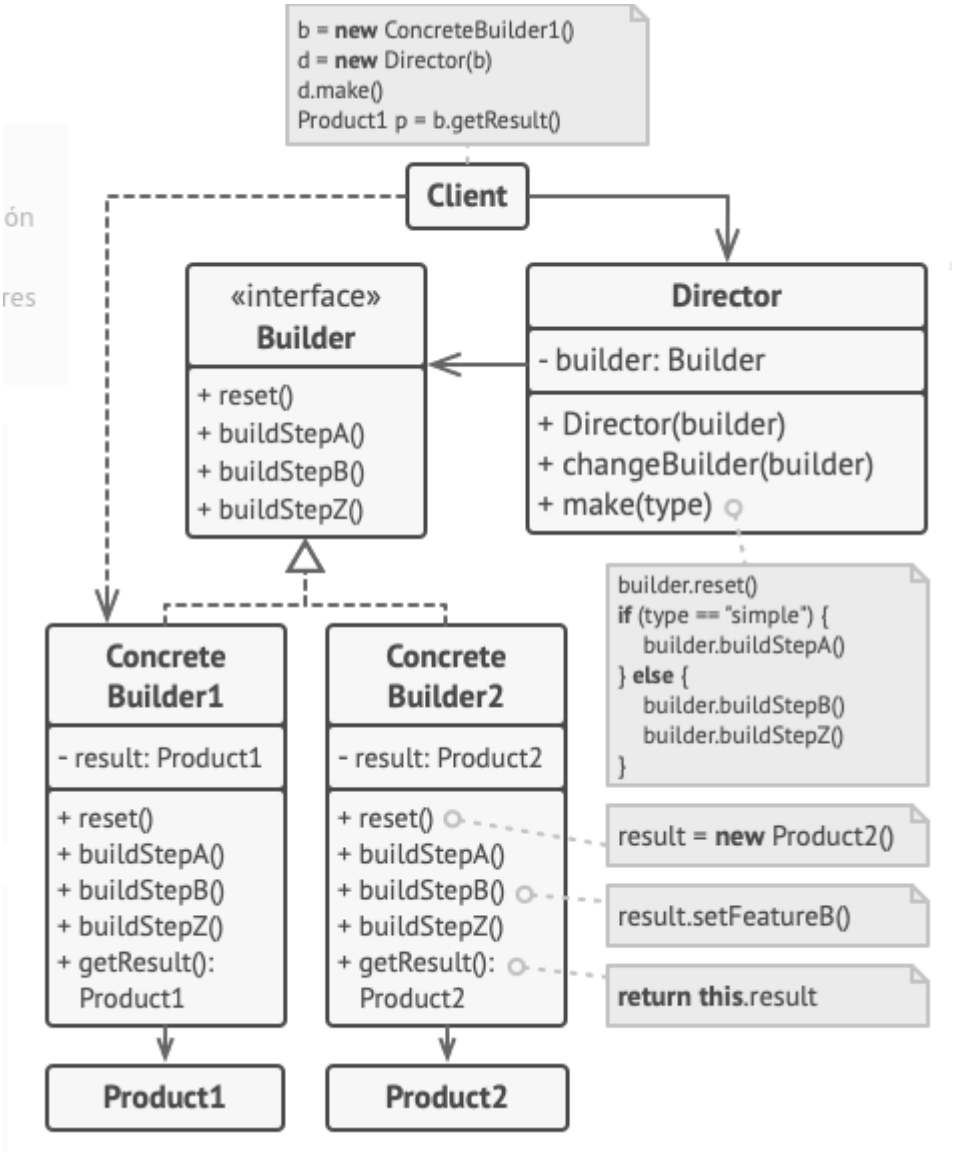
Los **productos concretos** son distintas implementaciones de la interfaz de producto

La clase **creador** declara el método factoría que devuelve nuevos objetos de producto. El tipo de retorno debe ser la interfaz de producto. Puede declararse abstracto para forzar a las subclases a implementarlo. También puede devolver un producto por defecto.

Los **creadores concretos** sobrescriben el método factoría base, de modo que devuelva un tipo diferente de producto

Patrón BUILDER (Constructor)

- **Propósito:** la construcción de un objeto complejo, separando el proceso de construcción de su representación, así que el mismo proceso puede crear diferentes representaciones.
- Es **aplicable** cuano deseamos crear un objeto complejo y que el algoritmo de creación sea independiente de las piezas que conforman el objeto complejo y de cómo se ensamblan. O cuando el proceso de construcción debe permitir diferentes representaciones para el objeto que se construye.
  - Para construir árboles con el patrón Composite u otros objetos complejos
- **Consecuencias:**
  - Permite cambiar la representación interna del producto
  - Reúne el código para la representación y el ensamblaje
  - Los clientes no necesitan saber nada sobre la estructura interna
  - Proporciona gran control del proceso de construcción
  - Diferentes directores pueden reutilizar un mismo builder
- Para **implementarlo**:
  - la interfaz de builder debe ser lo suficientemente general para permitir la construcción de productos para cualquier builder concreto
  - la construcción puede ser más complicada que añadir el nuevo token al producto en construcción
  - los métodos de la clase builder pueden no ser abstractos sino vacíos
  - las clases de los productos no siempre tienen una clase abstracta común
- "Considera un builder cuando te enfrentes a un constructor con muchos parámetros"
- El patrón BUILDER organiza la construcción de objetos en una serie de pasos. Para crear un objeto, se ejecuta una serie de estos pasos en un objeto builder. Lo importante es que no necesitas invocar todos los pasos, sino aquellos que sean necesarios para producir una configuración particular de un objeto.
- **Clase directora:** podemos extraer una serie de llamadas a los pasos del constructor que utilizamos para construir un producto y ponerlas en una clase independiente, la directora. Esta define el orden en el que se deben ejecutar los pasos de la construcción, mientras que el constructor proporciona la implementación de esos pasos. No es estrictamente necesaria, pero puede ser un buen lugar donde colocar distintas rutinas de construcción para poder reutilizarlas a lo largo del programa.
- **Estructura**



La interfaz **Builder** declara pasos de construcción de producto que todos los tipos de objetos constructores tienen en común

Los **builder concretos** ofrecen distintas implementaciones de los pasos de construcción. Pueden crear productos que no siguen la interfaz común

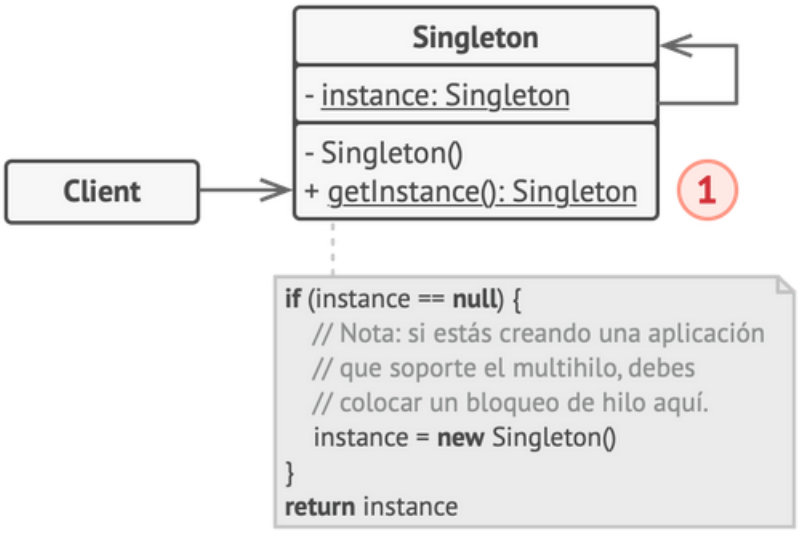
Los **productos** son los objetos resultantes. Los construidos por distintos objetos builder no tienen que pertenecer a la misma jerarquía de clases

La clase **directora** define el orden en el que se invocarán los pasos de construcción.

El **cliente** debe asociar uno de los objetos constructores con la clase directora

Patrón SINGLETON

- Su **propósito** es asegurar que una clase tiene una única instancia y es un punto de acceso global.
- Es **aplicable** cuando debe existir una única instancia de una clase, accesible globalmente.
- **Consecuencias:**
  - Acceso controlado a la única instancia
  - Evita usar variables globales
  - Es posible generalizar a un número variable de instancias
  - No es lo mismo que declarar todos los métodos estáticos (no es posible definir una jerarquía de clases y una clase puede cambiar y dejar de ser singleton)
  - La clase singleton puede tener subclases
  - Vulnera del principio de responsabiliad única
- Al usar singleton debemos ser cuidadosos al trabajar con **varios hilos**. La manera más sencilla de crear un singleton seguro ante concurrencia es hacer que el método de acceso global al singleton sea synchronized, de forma que un solo hilo puede ejecutar este método en un momento dado.
- La mejor forma de **implementar** un singleton es utilizar un tipo enumerado de un único elemento. Además, los singleton pueden ser inicializados de forma perezosa (se inicializan cuando se utilizan por primera vez) o temprana (se inicializan cuando se declaran).
- **Estructura**



La clase **singleton** declara el método estático *obtenerInstancia* que devuelve la misma instancia de su propia clase.

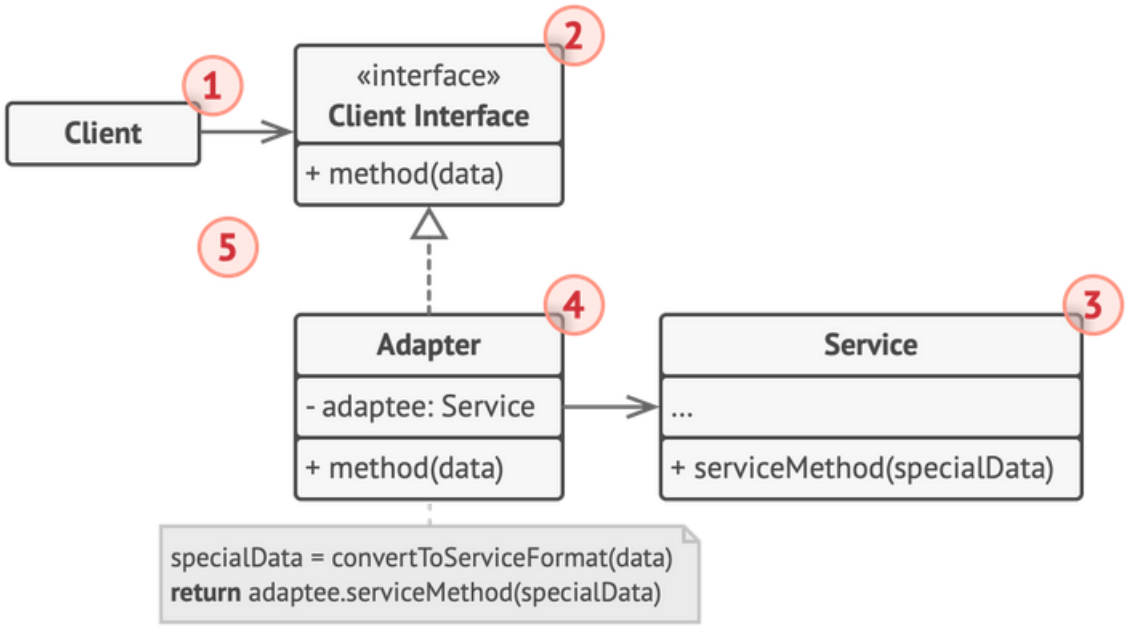
El constructor del singleton debe ocultarse al cliente. La llamada al método *obtenerInstancia* debe ser la única manera de obtener el objeto singleton.

2.2 Patrones Estructurales

Explican cómo ensamblar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura.

Patrón ADAPTER/WRAPPER (Adaptador)

- **Propósito:** convertir la interfaz de una clase en otra que una clase cliente espera. Permite la colaboración de ciertas clases a pesar de tener interfaces incompatibles.
- Es **aplicable** cuando se desea usar una clase existente y su interfaz no coincide con la que se necesita. O se desea crear una clase reutilizable que debe colaborar con clases no relacionadas o imprevistas.
- **Consecuencias:**
  - El tamaño de la clase adaptador depende de la similitud entre la interfaz de las clases objetivo y adaptado.
- **Funcionamiento:**
  1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes
  2. Usando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador
  3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en el formato y orden que ese segundo objeto espera
- **Estructura**



La clase **cliente** contiene la lógica de negocio existente del programa

La **interfaz con el cliente** describe un protocolo que otras clases deben seguir para poer colaborar con el código cliente

Servicio es alguna clase útil. El cliente no puede utilizarla directamente porque tiene una interfaz incompatible

La clase **adaptador** es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio

El código cliente no se acopla a la clase adaptador concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, puedes introducir nuevos tipos de adaptadores en el programa sin descomponer el código cliente existente.

Patrón DAO: Adapter + Abstract Factory + Singleton

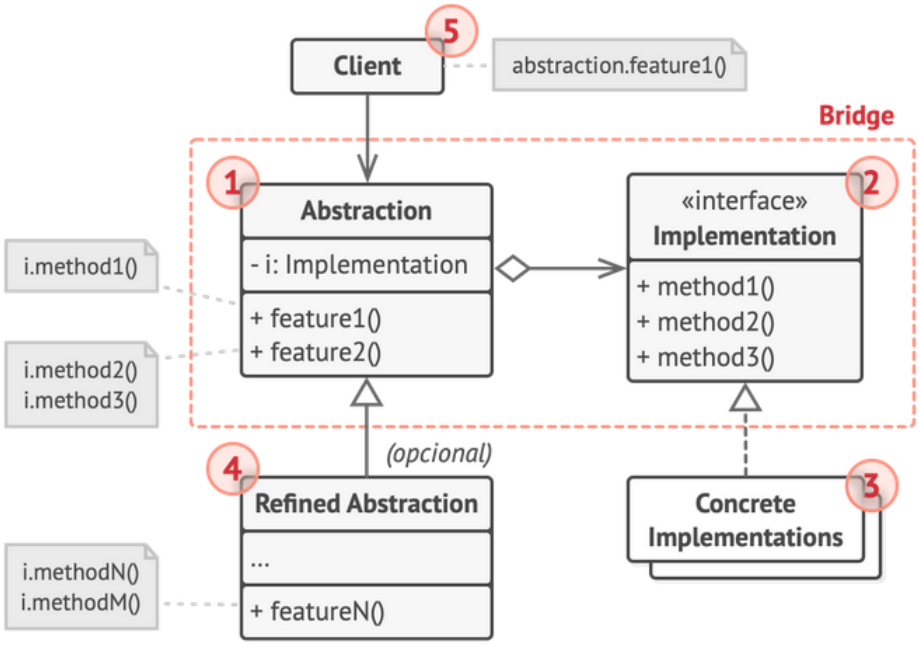
El patrón DAO (Data Access Object) se utiliza para conseguir que una aplicación sea independiente del sistema de almacenamiento utilizado. Para una clase de negocio se crea una interfaz DAO que proporciona los métodos CRUD para crear, recuperar, modificar y eliminar datos almacenados.

La interfaz DAO es implementada por clases que se encargan de establecer las conexiones a una fuente de datos concreta y encapsular cómo se accede a dicha fuente.

El patrón DAO utiliza una factoría abstracta para crear las instancias de las clases DAO que requiere la aplicación, para conseguir que esta sea independiente de un sistema concreto de almacenamiento.

Patrón BRIDGE/HANDLE

- Su **propósito** es desacoplar una abstracción de su implementación, de modo que los dos puedan cambiar independientemente. Permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (asbtracción + implementación) que pueden desarrollarse independientemente la una de la otra.
  - **Abstracción:** capa de control de alto nivel para una entidad. No hace ningún trabajo real por su cuenta, sino que delega el trabajo a la capa de **implementación**
- Viene **motivado** por situaciones en las que la herencia hace difícil reutilizar abstracciones e implementaciones de forma independiente y si refinamos la abstracción en una nueva subclase, esta tendrá tantas subclases como tenía la superclase.
- Respecto a la **implementación**
  - Debe hacerse el constructor de la clase Abstracción + el método setImplementación()
  - O elegir una implementación una implementación por defecto + el método setImplementación()
  - O delegar a otro objeto (por ejemplo factoría)
- Es **aplicable** si se quiere evitar una ligadura permanente entre una abstracción y su implementación, si las abstracciones e implementaciones son extensibles o si se tiene una proliferación de clases.
  - Queremos extender una clase en varias dimensiones independientes
  - Necesitamos poder cambiar implementaciones en tiempo de ejecución
- **Consecuencias:**
  - Un objeto puede cambiar su implementación en tiempo de ejecución
  - Se mejora la extensibilidad
  - Se ocultan detalles de implementación a los clientes
- **Estructura**



La **abstracción** ofrece lógica de control de alto nivel. Depende de que el objeto de la implementación haga el trabajo de bajo nivel

La **implementación** declara la interfaz común a todas las implementaciones concretas

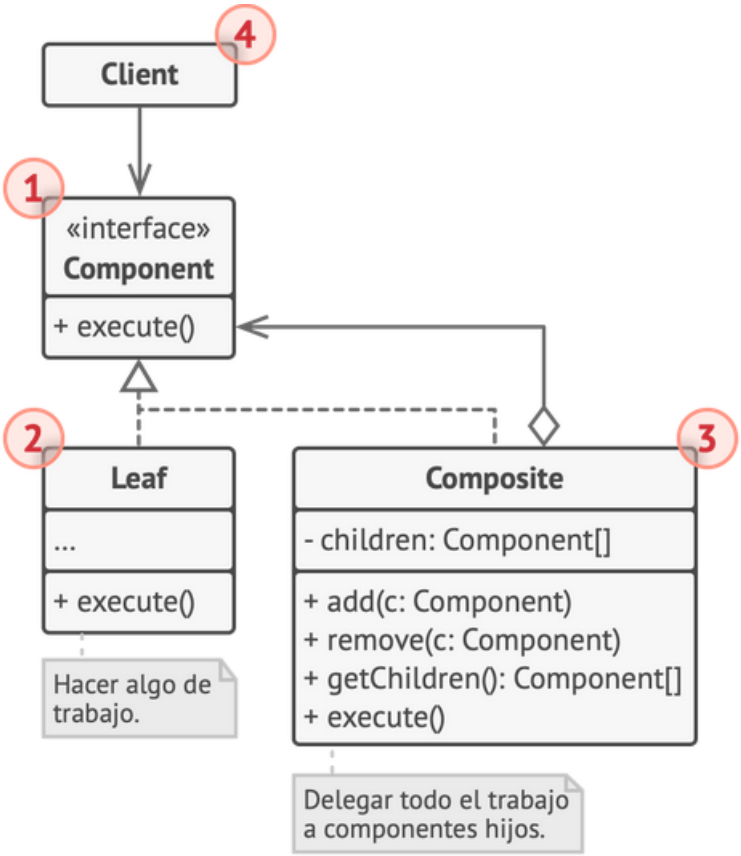
Las **implementaciones concretas** contienen código específico de plataforma

Las **abstracciones refinadas** proporcionan variantes de lógica de control

El **cliente** solo está interesado en trabajar con la abstracción. No obstante, el cliente tiene que vincular el objeto de la abstracción con uno de los objetos de la implementación

Patrón COMPOSITE

- **Propósito:** componer objetos en estructuras jerárquicas para representar jerarquías parte/todo. Permite al código cliente manejar a los objetos primitivos y compuestos de forma uniforme.
- Es **aplicable** cuando se quieren representar jerarquías parte/todo o se quiere que el cliente ignore la diferencia entre objetos completos y los objetos individuales que lo forman.
- **Consecuencias:**
  - Jerarquía con clases que modelan objetos primitivos y compuestos de modo uniforme
  - Clientes pueden tratar objetos primitivos y compuestos de modo uniforme
  - Es fácil añadir nuevos tipos de componentes
  - No se puede confiar al sistema de tipos que asegure que un objeto compuesto solo contendrá objetos de ciertas clases, por lo que es necesario comprobarlo en tiempo de ejecución.
- Para **implementarlo:**
  - las referencias de componentes hijos a su padre pueden ayudar al recorrido y manejo de la estructura compuesta
  - Operaciones de añadir, eliminar y recuperar hijos pueden estar en Componente Composite, llegando a un compromiso entre seguridad y transparencia
  - la estructura de datos adecuada para representar un composite dependerá del contexto
- **Estructura**



La interfaz **component** describe operaciones que son comunes a elementos simples y complejos del árbol

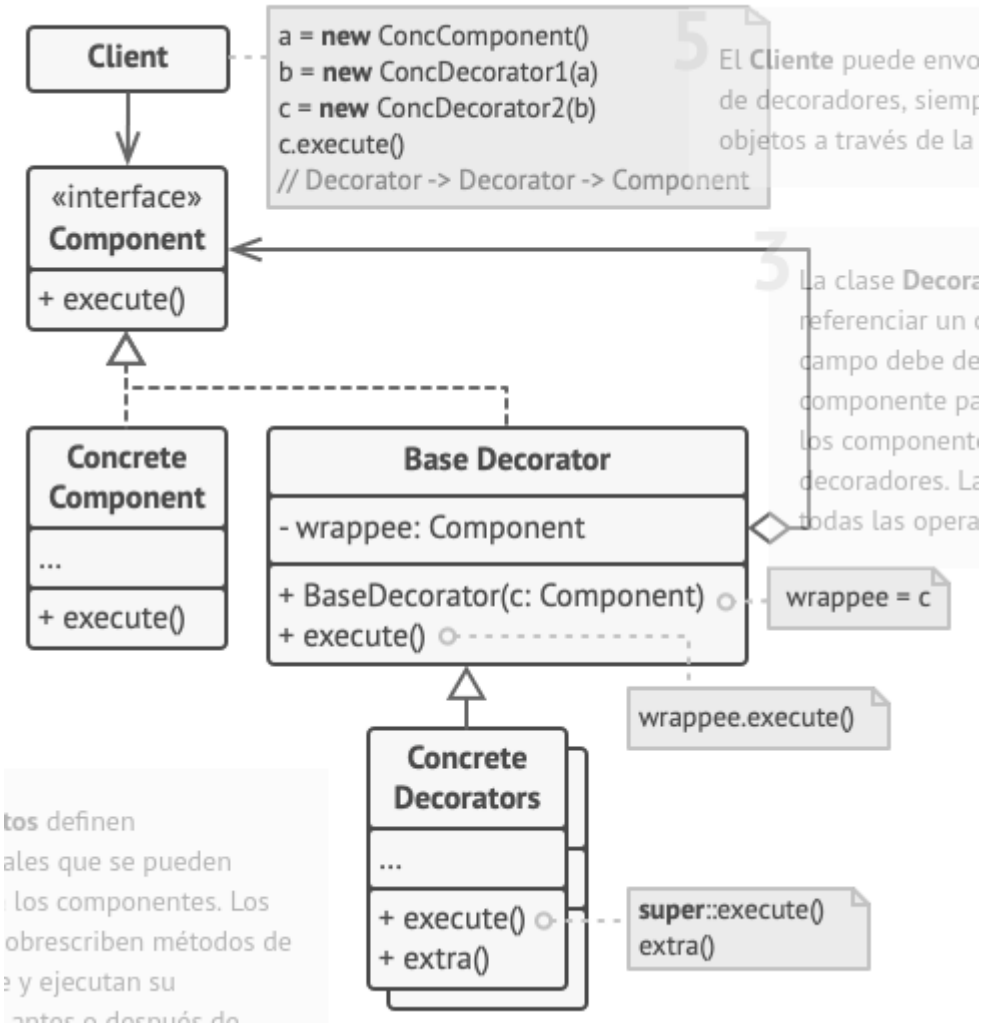
La **hoja** es un elemento básico de un árbol que no tiene subelementos

El **composite** es un elemento que tiene subelementos: hojas u otros composites. No conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente

El **cliente** funciona con todos los elementos a través de la interfaz componente. Así, el cliente puede funcionar de igual forma tanto con elementos simples como complejos

Patrón DECORATOR (Decorador)

- **Propósito:** asignar dinámicamente nuevas responsabilidades a un objeto. Es una alternativa más flexible a crear subclases para extender la funcionalidad de una clase.
- La **motivación** es que en ocasiones se desea añadir atributos o comportamiento adicional a un objeto concreto, no a una clase.
- Es **aplicable** cuando queremos añadir dinámicamente responsabilidades a objetos individuales de forma transparente, sin afectar a otros objetos o para evitar una explosión de clases
- **Consecuencias:**
  - más flexible que la herencia
  - diferentes decoradores pueden ser conectados a un mismo objeto
  - reduce el número de propiedades en las clases de la parte alta de la jerarquía
  - es simple añadir nuevos decoradores de forma independiente a las clases que extienden
  - puede dar lugar a aplicaciones con muchos y pequeños objetos
- **Implementación:**
  - los componentes y decoradores deben heredar de una clase común que debe ser ligera en funcionalidad. Si no es así, es mejor utilizar el patrón estrategia.
- **Estructura**



El **componente** declara la interfaz común tanto para wrappers como para objetos envueltos

El **componente concreto** es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar

La clase **decorador base** tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del compoennte, para que pueda contener tanto los componentes concretos como los decoradores. Esta clase delega todas las operaciones al objeto envuelto

Los **decoradores concretos** definen funcionalidad adicionales que se pueden añadir dinámicamente a los componentes. Sobrescriben métodos de la clase decoradores base y ejecutan su comportamiento, ya sea antes o despues de invocar al método padre

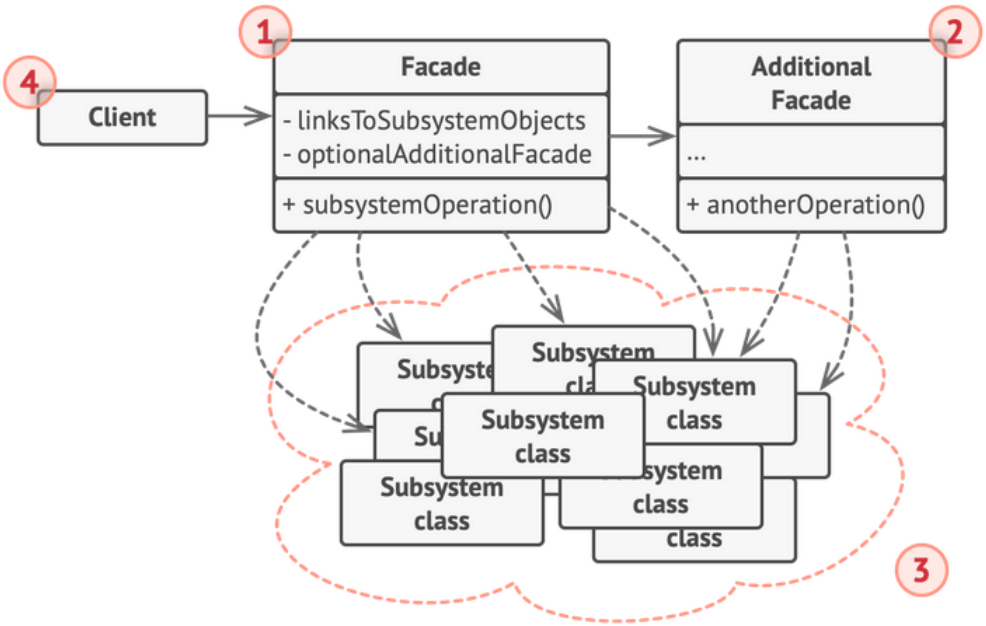
El **cliente** puede envolver componentes en varias capas de decoradores, siempre y cuando trabajen con todos los objetos a través de la interfaz del componente

- Diferencias entre Decorador y Estrategia:

Decorador	Estrategia
Un componente no sabe nada sobre sus decoradores	Una estrategia no sabe nada sobre sus componentes
El decorador envía los mensajes al componente que decora, pudiendo extender la operación con nuevo comportamiento	La estrategia solo define un comportamiento del componente, en función del contexto que este le proporciona
Las clases que modelan los decoradores pueden añadir responsabilidades a las que heredan de la clase Decorador	

Patrón FACADE (Fachada)

- Su **propósito** es proporcionar una única interfaz a un conjunto de clases de un subsistema que ofrece una funcionalidad. Define una interfaz de más alto nivel que facilita el uso de un subsistema.
- **Motivación:** reducir las dependencias entre subsistemas
- Es **aplicable** cuando deseamos proporcionar una interfaz simple a un subsistema, hay muchas dependencias entre clientes y las clases que implementan una abstracción o se desea una arquitectura de varios niveles: una fachada define el punto de entrada para cada nivel-subsistema
- **Consecuencias:**
  - facilita a los clientes el uso de un subsistema al ocultar sus componentes
  - proporciona un acoplamiento débil entre un subsistema y los clientes: cambios en los componentes no afectan a los clientes
  - no se impide a los clientes el uso de las clases del subsistema si lo necesitan
- **Implementación:**
  - es posible reducir el acoplamiento entre clientes y el subsistema definiendo la fachada como una clase abstracta con una subclase por cada implementación del subsistema.
  - la fachada no es la única parte pública de un subsistema, sino que es posible declarar clases individuales del subsistema como públicas
  - una fachada es normalmente un singleton
- **Estructura**



El patrón **facade** proporciona un práctico acceso a una parte específica de la funcionalidad del subsistema. Sabe a dónde dirigir la petición del cliente y cómo operar todas las partes móviles

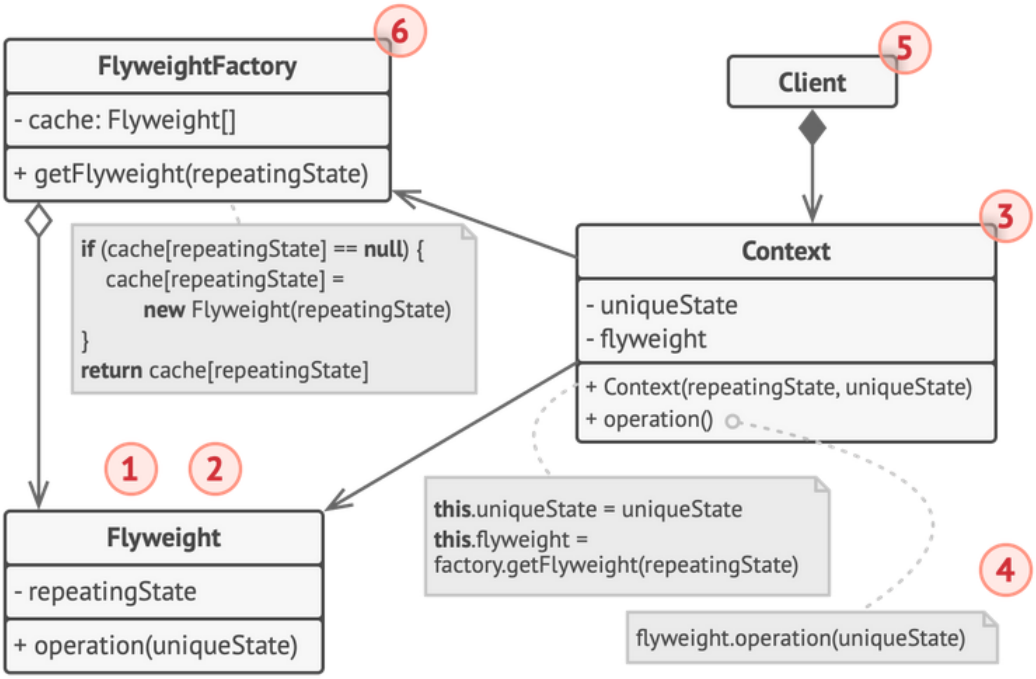
Puede crearse una clase **fachada adicional** para evitar contaminar una única fachada con funciones no relacionadas que podrían convertirla en otra estructura compleja

El **subsistema complejo** consiste en decenas de objetos diversos. Para lograr que todos hagan algo significativo, debemos profundizar en los detalles de implementación del subsistema, que pueden incluir inicializar objetos en un orden concreto y suministrarles datos en un formato dado. Las clases del subsistema no conocen la existencia de la fachada, operan dentro del sistema y trabajan entre sí directamente

El **cliente** utiliza la fachada en lugar de invocar directamente los objetos del subsistema

Patrón FLYWEIGHT (Peso Ligero)

- **Propósito:** uso de objetos compartidos para soportar eficientemente un gran número de objetos de poco tamaño.
- **Motivación:** un flyweight es un objeto compartido que puede ser utilizado en diferentes contextos simultáneamente, pues no hace asunciones sobre el contexto. Se utilizan para modelar conceptos o entidades de los que se necesita una gran cantidad en una aplicación.
- Presenta dos estados:
  - **estado intrínseco:** se almacena en el flyweight y consiste de información que es independiente del contexto y se puede compartir
  - **estado extrínseco:** depende del contexto, por lo que no puede ser compartido. Son los objetos clientes los que tienen esta información.
- Es **aplicable** cuando se cumplen las siguientes condiciones:
  - una aplicación utiliza un gran número de objetos de cierto tipo
  - el coste de almacenamiento es alto debido al excesivo número de objetos
  - la mayor parte del estado de esos objetos puede hacerse extrínseco
  - al separar el estado extrínseco, muchos grupos de objetos pueden reemplazarse por unos pocos objetos compartidos
  - la aplicación no depende de la identidad de los objetos
- **Consecuencias:**
  - puede introducir costes run-time debido a la necesidad de calcular y transferir el estado extrínseco
  - la ganancia de espacio depende de la reducción del número de instancias, del tamaño del estado extrínseco por objeto y de si el estado extrínseco es almacenado o calculado
  - interesa que el estado extrínseco sea calculado
- **Implementación:** debido a que los flyweight son compartidos, no deberían ser instanciados directamente por los clientes, es recomendable el uso de una factoría
- **Estructura**



El patrón FLYWEIGHT es simplemente una optimización. Antes de aplicarlo, debemos asegurarnos de que nuestro problema tiene un problema de consumo de RAM provocado por tener una gran cantidad de objetos similares en memoria al mismo tiempo

La clase **Flyweight** contiene la parte del estado del objeto original que pueden compartir varios objetos. El mismo objeto flyweight puede utilizarse en muchos contextos diferentes. El estado almacenado dentro de un objeto flywight es el intrínseco, mientras que el que se pasa a sus métodos es el extrínseco

La clase **contexto** contiene el estado extrínseco, único en todos los objetos originales. Cuando un contexto se empareja con uno de los objetos flyweight, representa el estado completo del objeto original

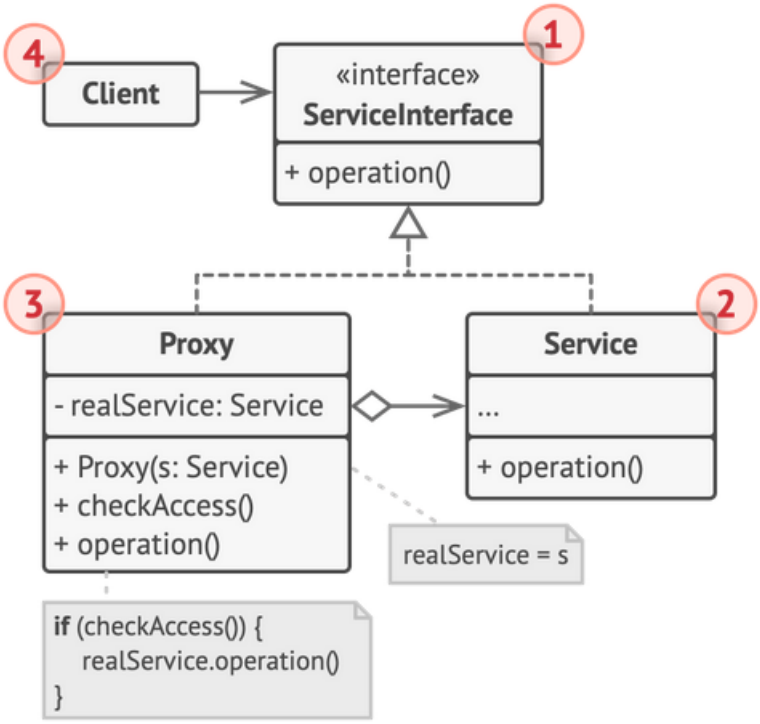
El comportamiento del objeto original permanece en la clase flyweight. Quien invoque un método del objeto flyweight debe también pasar las partes adecuadas del estado extrínseco dentro de los parámetros del método

El **cliente** calcula o almacena el estado extrínseco de los objetos flyweight

La **fábrica flyweight** gestiona un grupo de objetos flyweight existentes. Con la fábrica, los clientes no crean objetos flyweight directamente. En lugar de eso, invocan a la fábrica, pasándole partes del estado intrínseco del objeto flyweight deseado. La fábrica revisa objetos flyweight creados previamente y devuelve uno existente que coincida con los criterios de búsqeuda o bien crea uno nuevo si no hay coincidencias

Patrón PROXY (Sustituto)

- **Propósito:** proporcionar un sustituto de un objeto para controlar el acceso a dicho objeto
- **Motivación:**
  - diferir el coste de crear un objeto hasta que sea necesario usarlo: creación bajo demanda
  - hay situaciones en las que un objeto cliente no referencia o no puede referenciar a otro objeto directamente, pero necesita interactuar con él. Un objeto proxy puede actuar como intermediario entre el objeto cliente y el objeto destino
- Para **implementarlo** se convierte una referencia en un objeto que ocnforma con el objeto sustituido. El objeto proxy tiene la misma interfaz que el objeto destino. El objeto proxy mantiene una referencia al objeto destino y puede pasarle a él los mensajes recibidos.
- Es **aplicable** cuando hay necesidad de referencias a un objeto mediante una referencia más rica que un puntero o una referencia normal (como un proxy de acceso remoto, virtual, para protección o de referencia inteligente, este último proporciona operaciones adicionales a las que proporciona el propio objeto)
- **Consecuencias:** introduce un nivel de indirección para:
  - un proxy remoto oculta el hecho de que los objetos residen en diferentes espacios de direcciones
  - un proxy virtual puede crear o copiar un objeto bajo demanda. Se usa cuando tenemos un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesitemos de vez en cuando
  - un proxy para protección o las referencias inteligentes permiten realizar tareas de control sobre los objetos accedidos
- **Proxy para clonación perezosa:** una razón para clonar un objeto Map es evitar mantener un bloqueo sobre la colleción un largo tiempo, si solo se desean realizar operaciones de consulta. Los métodos synchronized para obtener el acceso exclusivo pueden resultar inaceptables en algunas situaciones. Algunas clases que implementan Map permiten la clonación, pero una clonación previa puede ser innnecesaria, es mejor aplicar una clonación perezosa y solo clonar cuando sea necesario.
- **Estructura**



La **interfaz del servicio** declara la interfaz del servicio. El proxy debe seguir esta interfaz para poder camuflarse como objeto de servicio

Servicio es una clase que proporciona una lógica de negocio

La clase **proxy** tiene un campo de referencia que apunta a un objeto de servicio. Cuando el proxy finaliza su procesamiento, pasa la solicitud al objeto de servicio

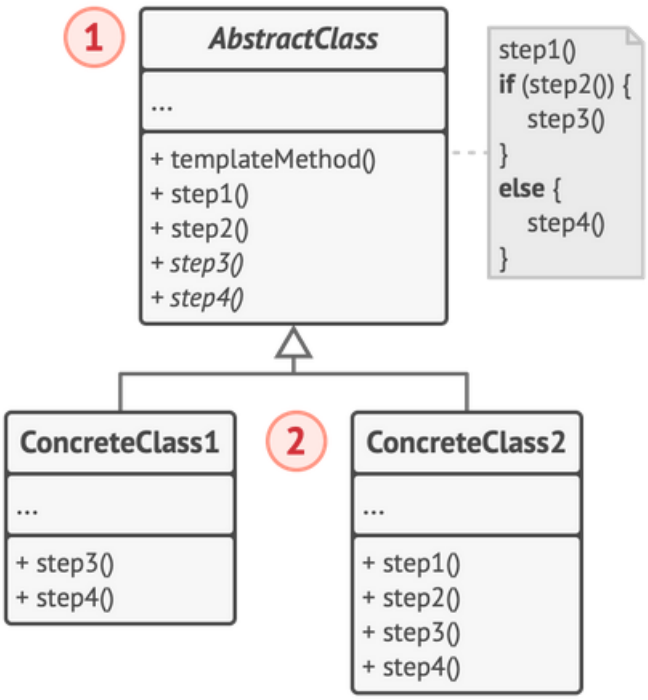
El **cliente** debe funcionar con servicios y proxies a través de la misma interfaz. De este modo podemos pasar un proxy a cualquier código que espere un objeto de servicio

2.3 Patrones de Comportamiento

Tratan con algoritmos y la asignación de responsabilidades entre objetos

Patrón TEMPLATE METHOD (Método Plantilla)

- Su **propósito** es definir operaciones como esquemas de algoritmos que difieren algunos pasos a operaciones implementadas en las subclases. Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.
- **Motivación:** es esencial para la reutilización en jerarquías de clases: factorizar comportamiento común.
- Es **aplicable** cuando una clase implementa el esquema de un algoritmo y deja que las subclases implementen el comportamiento que puede variar o cuando el comportamiento común entre varias clases debe ser factorizado y localizado en una superclase común.
- **Consecuencias:** un método plantilla invoca a los siguientes tipos de métodos:
  - operaciones abstractas
  - operaciones concretas en la clase abstracta
  - operaciones concretas en clientes
  - métodos factoría
- **Estructura**

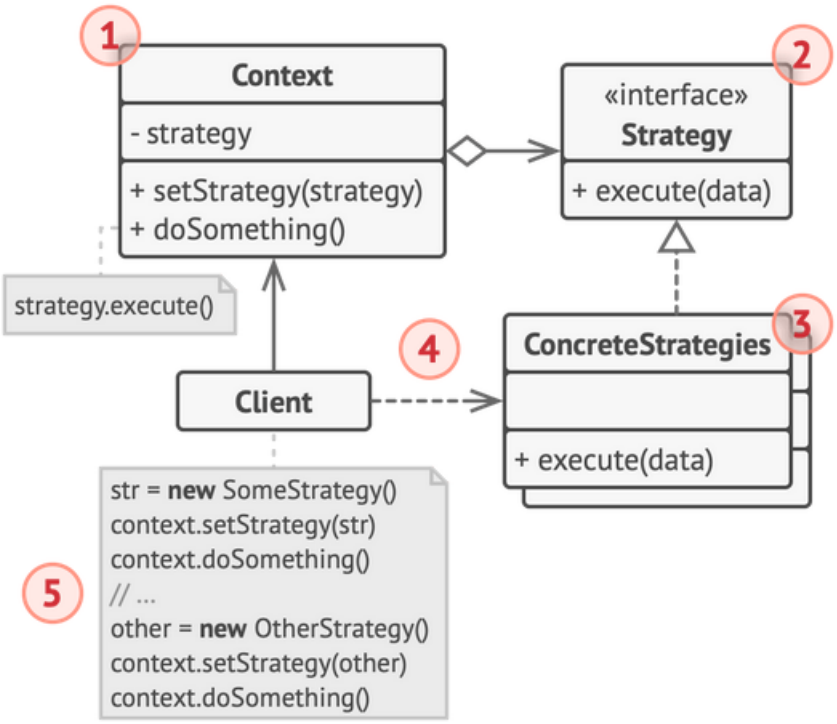


La **clase abstracta** declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse abstractos o con una implementación por defecto

Las **clases concretas** pueden sobrescribir todos los pasos, pero no el propio método plantilla

Patrón STRATEGY/POLICY (Estrategia)

- **Propósito:** define una familia de algoritmos, encapsula cada uno, y permite intercambiarlos. Permite variar los algoritmos de forma independiente a los clientes que los usan.
- **Implementación:**
  - ¿Cómo una estrategia concreta accede a los datos del contexto? Se le pueden pasar datos o un objeto contexto como argumento, o estrategia almacena una referencia al contexto
  - ¿Cómo se crea una instancia de una estrategia concreta? Mediante el uso de una factoría.
- Es **aplicable** cuando queremos configurar una clase con uno de varios comportamientos posibles, o se necesitan diferentes variantes de un algoritmo, o una clase define muchos comportamientos que aparecen como sentencias case en sus métodos.
- **Consecuencias:**
  - define una familia de algoritmos relacionados
  - una alternativa a crear subclases de la clase contexto
  - elimina sentencias case
  - en el código fuente se puede elegir entre diferentes estrategias o implementaciones: debe conocer detalles
  - STATE y STRATEGY son similares, pero cambia el propósito
- **Estructura**



La clase **contexto** mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia

La interfaz **estrategia** es común a todas las estrategias concretas. Declara un método que la clase contexto utiliza para ejecutar una estrategia

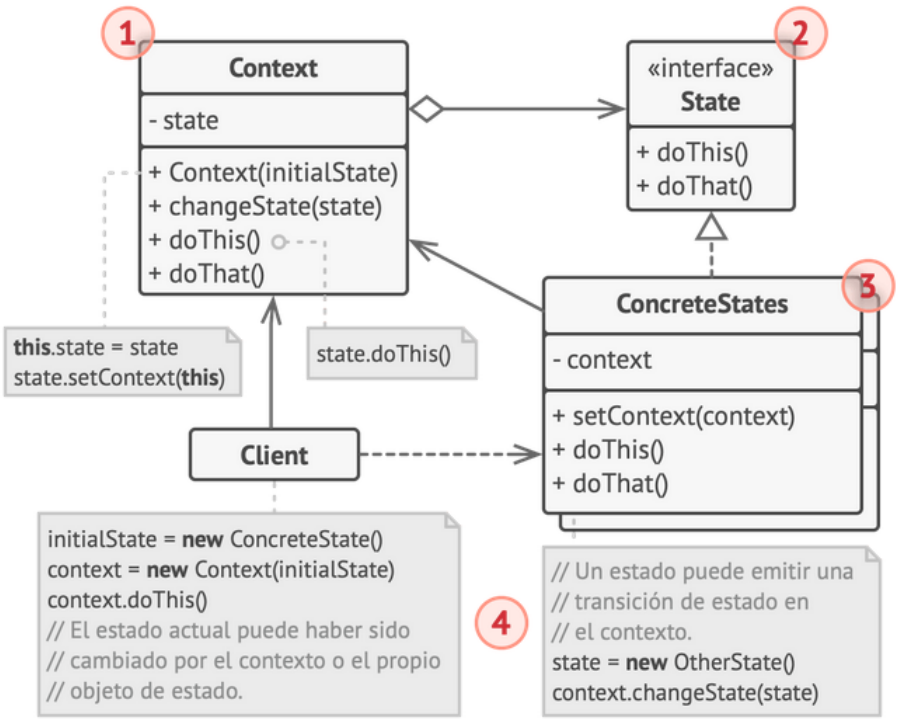
Las **estrategias concretas** implementan distintas variaciones de un algoritmo que la clase contexto utiliza

La clase contexto invoca al método de ejecución en el objeto de estrategia vinculado cada vez que necesita ejecutar el algoritmo. La clase contexto no sabe con qué tipo de estrategia funciona o cómo se ejecuta el algoritmo

El **cliente** crea un objeto de estrategia específico y lo pasa a la clase contexto, esta tiene un modificador *set* que permite a los clientes sustituir la estrategia asociada al contexto en tiempo de ejecución

Patrón STATE (Estado)

- **Propósito:** permite a un objeto cambiar su comportamiento cuando cambia su estado. El objeto parece cambiar de clase.
- **Implementación:**
  - ¿Quién define las transiciones entre estados? El contexto o las subclases estado. Más apropiado que sean los estados.
  - ¿Cuándo son creados los objetos estado? Pueden crearse cuando se necesiten o con antelación y que contexto tenga una referencia a ellos.
- Es **aplicable** cuando el comportamiento del objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de su estado o las operaciones tienen grandes estructuras case que dependen del estado del objeto, que es representado por uno o más constantes de tipo enumerado.
- **Consecuencias:**
  - Todo el comportamiento asociado a un particular estado es embebido de una clase
  - Subclases en vez de sentencias case
  - Ayuda a evitar estados inconsistentes en caso de estado representado por varias variables
  - Transiciones de estado son más explícitas
  - Incrementa el número de objetos. Los objetos estado pueden ser singleton.
- **Estructura**



La clase **contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. Se comunica con el estado a través de la interfaz de estado. Tiene un modificador *set* para pasarle un nuevo objeto de estado.

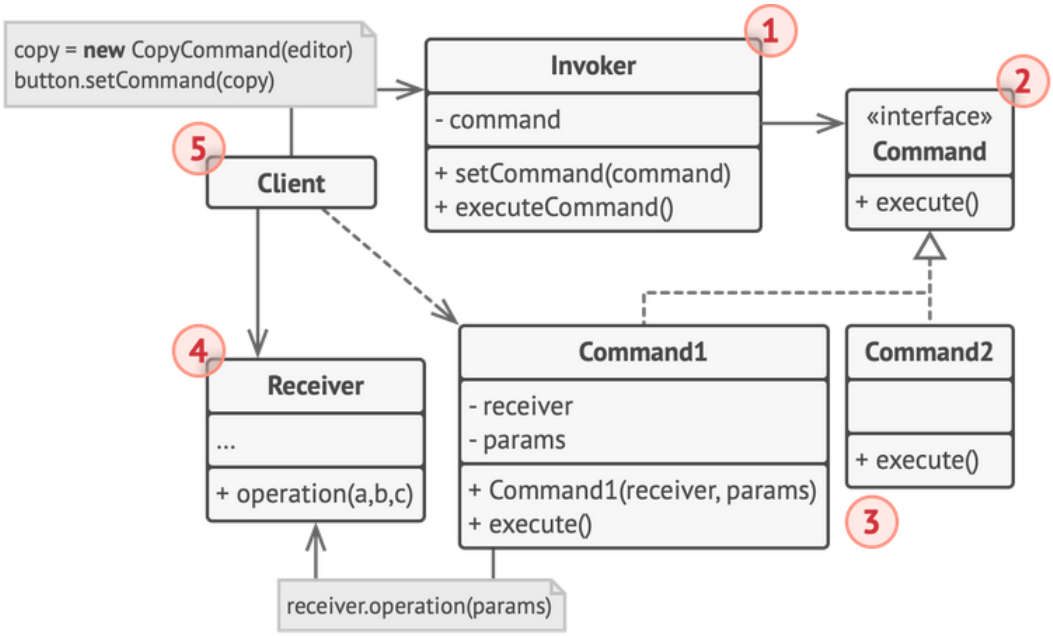
La interfaz **estado** declara los métodos específicos del estado. Estos deben tener sentido para todos los estados concretos

Los **estados concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia el estado puede extraer cualquier información requerida del objeto de contexto y hacer transiciones de estado

Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transicicón de estado sustituyendo el objeto de estado vinculado al contexto

Patrón COMMAND (Orden)

- **Propósito:** encapsular un mensaje como un objeto, permitiendo parametrizar métodos y objetos como mensajes, añadir mensajes a una cola y soportar funcionalidad deshacer/rehacer.
- La **motivación** es que algunas veces es necesario enviar un mensaje a un objeto sin conocer el selector del mensaje ni el objeto receptor.
- **Implementación:** en lenguajes que soportan expresiones lambda, punteros a funciones o introspección, el patrón COMMAND solo es útil para soportar undo/redo.
- **Aplicabilidad:**
  - Para parametrizar objetos por la acción a realizar cuando el lenguaje no soporta pasar código como parámetro de métodos
  - Añadir a una cola y ejecutar mensajes después de su ejecución. Un objeto Command tiene un tiempo de vida independiente de la solicitud original. Para soportar undo/redo. Y para recuperación de fallos.
- **Consecuencias:**
  - desacopla el objeto que invoca la operación del objeto que sabe cómo realizarla
  - cada subclase CommandConcreto encapsula un par receptor/acción, almacenando el receptro como un atributo e implementando el método ejecutar
  - objetos Command pueden ser manipulados como cualquier otro objeto
  - Se pueden crear command compuestos aplicando el patrón COMPOSITE
- **Mecanismo Undo/Redo:** una jerarquía de Commands representa las operaciones que se pueden deshacer/rehacer. Los Command deben tener atributos para registrar la información que permita deshacer/rehacer. Hay una lista historia de objetos Command que registra copias de las operaciones realizadas (una única lista doblemente enlazada, o dos listas historia y redo). Una clase manager soporta el manejo de las listas de Commands y de la recepción de operaciones. Las operaciones undo y redo se tratan de forma distinta al resto de commands.
- **Estructura**



La clase **invocador** es responsable de inicializar las solicitudes. Esta clase debe tener un campo para almacenar una referencia a un objeto de comando. El emisor activa este comando en lugar de enviar la solicitud directamente al receptor. El invocador no es responsable de crear el objeto de comando. Normalmente, obtiene un comando precreado de parte del cliente a través del constructor

La interfaz **comando** normalmente declara un único método para ejecutar el comando

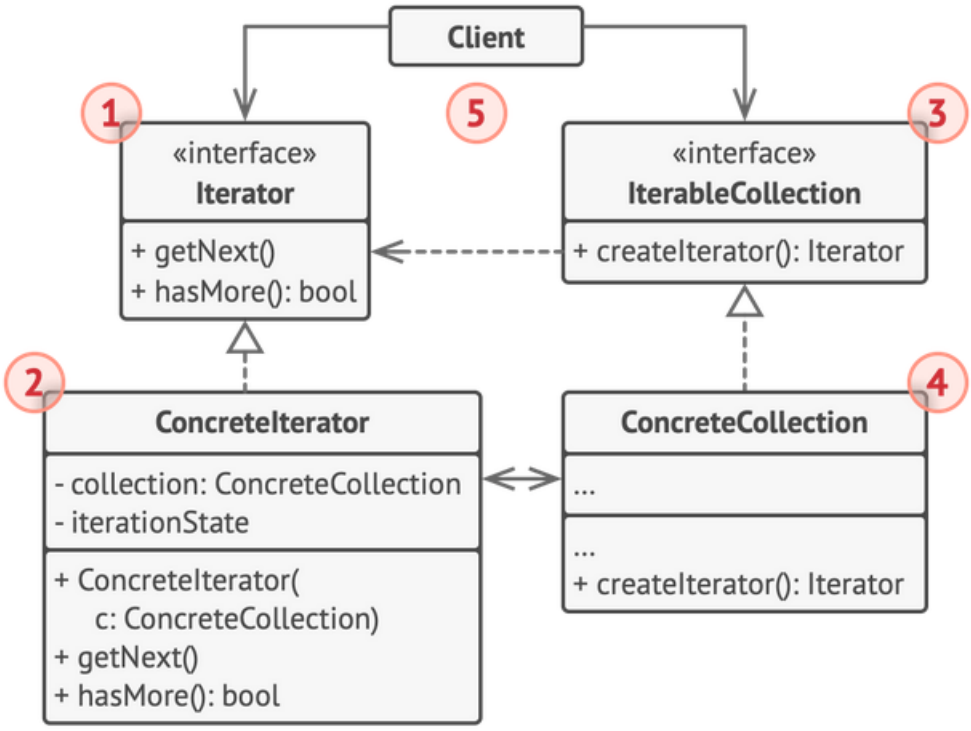
Los **comandos concretos** implementan varios tipos de solicitudes. Un comando concreto no se supone que tenga que realizar el trabajo por su cuenta, sino pasar la llamada a uno de los objetos de la lógica de negocio. Los parámetros necesarios para ejecutar un método en un objeto receptor pueden declararse como campos en el comando concreto

La clase **receptor** contiene cierta lógica de negocio. Casi cualquier objeto puede actuar como receptor. La mayoría de los comandos solo gestiona los detalles sobre cómo se pasa una solicitud al receptor, mientras que es este el que hace el trabajo real

El **cliente** crea y configura los objetos de comando concretos. Este debe pasar todos los parámetros de la solicitud, incluyendo una instancia del receptor, dentro del constructor del comando

Patrón ITERATOR (Iterador)

- **Propósito:** proporciona una forma para acceder a los elementos de una estructura de datos sin exponer los detalles de la representación.
- **Motivación:**
  - un objeto contenedor debe permitir una forma de recorrer sus elementos sin exponer su estructura interna
  - Debería permitir diferentes métodos de recorrido y recorridos concurrentes
  - Esta funcionalidad no es parte de la interfaz de la colección
  - Los iteradores pueden ser **externos** (recorrido controlado por el código que usa el iterador, tarea del programador) o **internos** (recorrido controlado por el propio interador, se libera al programador que solo dice lo que quiere)
- **Consecuencias:**
  - simplifica la interfaz de una colección al extraer los métodos de recorrido
  - permite varios recorridos concurrentes
  - soporta variantes en las técnicas de recorrido
- **Implementación**
  - ¿Quién controla la iteración? Externos VS Internos
  - ¿Quién define el algoritmo de recorrido? **Agregado:** el iterador solo almacena el estado de la iteración. **Iterador:** es posible reutilizar el mismo algoritmo sobre diferentes colecciones o aplucar diferentes algoritmos sobre una misma colección
  - ¿Es posible modificar la colección durante la iteración?
  - Colección e iterador son clases muy relacionadas
  - Puede ser usado junto a patrón composite
- **Estructura**



La interfaz **iterador** declara las operaciones necesarias para recorrer una colección: estraer el siguiente elemento, recuperar la posición actual, reiniciar la iteración,...

Los **iteradores concretos** implementan algoritmos específicos para recorrer una colección. El objeto iteradir debe controlar el progreso del recorrido por su cuenta, así puede haber varios iteradores simultáneamente sobre la misma colección

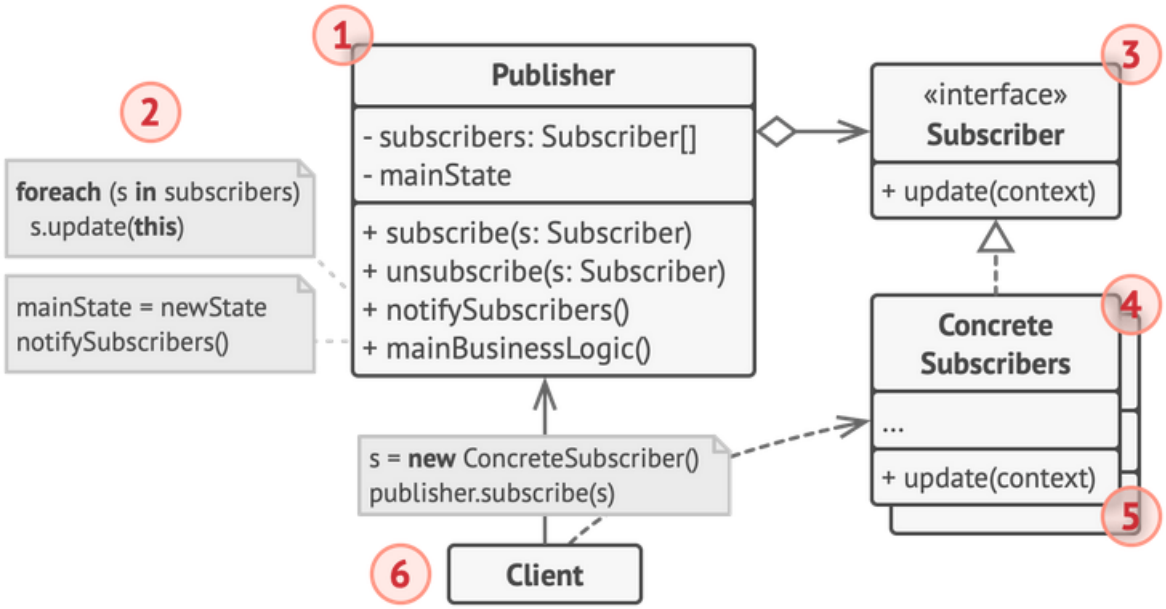
La interfaz **colección** declara uno o varios métodos para obtener iteradores compatibles con la colección. El tipo de retorno de los métodos debe declararse como la interfaz iteradora de forma que las colecciones concretas puedan devolver varios tipos de iteradores

Las **colecciones concretas** devuelven nuevas instancias de una clase iteradora concreta particular cada vez que el cliente solicita una

El **cliente** debe funcionar con colecciones e iteradores a través de sus interfaces. De este modo, el cliente no se acopla a clases concretas, permitiendo usar varias colecciones e iteradores con el mismo código cliente

Patrón OBSERVER / PUBLISH-SUBSCRIBE (Observador)

- **Propósito:** define una dependencia uno a muchos entre objetos, de modo que cuando cambia el estado de un objeto, todos sus dependientes automáticamente son notificados
- **Aplicabilidad:** cuando un cambio de estado en un objeto requiere cambios en otros objetos, y no sabe sobre qué objetos debe aplicarlos o cuando un objeto debe ser capaz de notificar algo a otros objetos, sin hacer asunciones sobre quiénes son estos objetos
- **Consecuencias:**
  - acoplamiento abstracto y mínimo entre el sujeto y el observador, para poder reutilizarlos por separado, puedan estar en diferentes capas, puedan añadirse observers sin modificar el subject, y el subject no necesite concoer las clases concretas de observers
  - es posible añadir y eliminar observers en cualquier instante
- **Implementación:**
  - es posible que un observador esté ligado a más de un sujeto: la operación update tendrá como argumento el sujeto
  - al registrar un observador es posible asociarle el evento sobre el que quiere ser notificado
  - ¿Quién dispara la notificación? Normalmente, métodos set en la clase Subject, en vez de clases clientes de la clase Subject
  - ¿Cuánta información sobre el cambio se le envía a los observers con la notificación? Conveniente pasarle el subject o un objeto
- **Estructura**



El **notificador** envía eventos de interés a otros objetos. Esos eventos cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los notificadores contienen una estructura de suscripción que permite a nuevos y antiguos suscriptores abandonar la lista

Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor

La interfaz **suscriptor** declara la interfaz de notificación. En la mayoria de casos consiste en un único método *update*

Los **suscriptores concretos** realizan algunas acciones en resppuesta a las notificaciones emitidas por el notificador. Todas estas clases deben implementar la misma interfaz, de forma que el notificador no esté acoplado a clases concretas

Los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por esto, a menudo los notificadores pasan cierta información de contexto como argumento del método *update*

El **cliente** crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las actualizaciones del notificador

- **MODELO DE DELEGACIÓN DE EVENTOS (MDE):** modelo de eventos basado en el patrón Observer.
  - Los objetos que pueden generar eventos son llamados fuentes de eventos
  - Los objetos que desean ser notificados de eventos son llamados oyentes de eventos (event listeners)
  - Los listeners deben registrarse en las fuentes e implementan una interfaz con los métodos que deben ser llamados por la fuente cuando ocurre el evento.

3 Interfaces funcionales

Las **interfaces funcionales** son interfaces con un único método que es utilizada como tipo de una expresión lambda. Las **expresiones lambda** son bloques de código compatibles con una interfaz funcional.

- **Interfaz Predicate<T>:**
  - comprueba la condición sobre el argumento dado
  - ```
    public interface Predicate<T> {
            boolean test(T t);
    }
    ```
  - Ejemplo:

    ```
    public class testEq implements Predicate<String> {
            private String miString;

            testEq(String s) miString = s;

            public boolean test(String s){
                    return miString.equals(s);
            }
    }

    //para aplicarlo

    String palabraEspecial;
    testEq eq = new testEq(palabraEspecial);
    for (String s : palabras)
            if (eq.test(s)) print(s);

    //equivalentemente

    palabras.stream().filter(s -> s.equals(palabraEspecial))
                    .forEach(s -> print(s));
    ```

- **Interfaz Consumer<T>:**
  - representa operaciones que dado un único argumento realizan una acción y no retornan nada
  - ```
    public interface Consumer<T> {
            void accept(T t);
    }
    ```
  - Ejemplo:

    ```
    public class printFloor implements Consumer<Double> {

            printFloor(){}

            public void accept(Double d){
    ```

```
                print(d.floor());
        }
}

//para aplicarlo

Double num;
printFloor eq = new printFloor();
for (Double d : numeros)
        if (d<=num) eq.accept(d);

//equivalentemente

numeros.stream().filter(d -> d<=num).forEach(d -> print(d.floor()));
```

- **Interfaz Supplier<T>:**
  - Representa un suministrador de resultados
  - ```
    public interface Supplier<T> {
            T get();
    }
    ```
  - Ejemplo:

    ```
    public class randomNumber implements Supplier<Double> {
            Random rand;
            Double min, max;
            testEq(double m, double M) {
                    min = m;
                    max = M;
                    rand = new Random();
            }

            public Double get(){
                    return rand.nextDouble(max-min)-min;
            }
    }

    //para aplicarlo

    randomNumber r = new randomNumber(-5.0,5.0);
    for (Double d : numeros)
            if (r.get()>0) print(d);

    //equivalentemente

    numeros.stream().forEach(d -> if(r.get()>0) print(d));
    ```

- **Interfaz Function<T,R>:**
  - representa una función que acepta un argumento y devuelve un resultado
    ```

- ```
  public interface Function<T,R> {
          R apply(T t);
  }
  ```

- Ejemplo:

  ```
  public class getAutor implements Function<Libro, Autor> {

          getAutor() {}

          public Autor apply(Libro l){
                  return l.getAutor();
          }
  }

  //para aplicarlo

  getAutor eq = new getAutor();
  for (Libro l : biblioteca)
          if (getAutor(l).equals("Cervantes")) print("Found!");

  //equivalentemente

  biblioteca.stream().map(l -> l.getAutor())
                  .filter(s -> s.equals("Cervantes"))
                  .forEach(s -> print("Found!");
  ```
  ```