SOLID: Principios de diseño (POO).

- SRP → SINGLE-RESPONSIBILITY PRINCIPLE: Nunca debe haber más de una razón para que una clase cambie. Agrupa las cosas que cambian por las mismas razones. Separa las cosas que cambian por razones diferentes (diferentes roles). La razón por la que es importante mantener una clase enfocada en una sola preocupación es que hace que la clase sea más robusta.

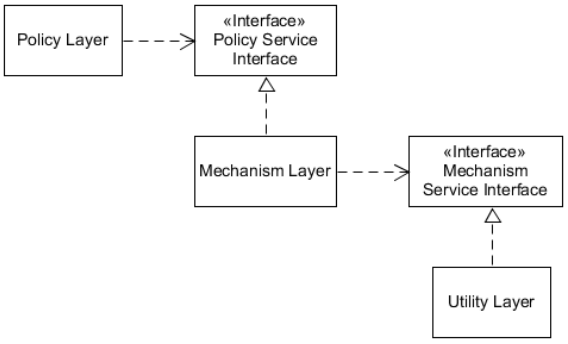
- OCP → OPEN-CLOSED PRINCIPLE: Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificación. Una entidad puede permitir que su comportamiento sea extendido sin modificar su código fuente (Generalización-Herencia). Se dirá que un módulo está cerrado si está disponible para su uso por otros módulos. Esto asume que al módulo se le ha dado una descripción bien definida y estable.
    - Polymorphic open–closed principle: (Clases abstractas-Interfaces) las implementaciones pueden cambiarse y se pueden crear múltiples implementaciones que podrían ser sustituidas polimórficamente unas por otras.

- LSP → LISKOV SUBSTITUTION PRINCIPLE: Las funciones que usan punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo. Un objeto (como una clase) puede ser reemplazado por un subobjeto (como una clase que extiende la primera clase) sin romper el programa.

- ISP → INTERFACE SEGREGATION PRINCIPLE: Ningún código debe ser forzado a depender de métodos que no usa. ISP divide interfaces muy grandes en interfaces más pequeñas y específicas para que los clientes solo tengan que conocer los métodos que les interesan. El ISP está destinado a mantener un sistema desacoplado y, por lo tanto, más fácil de refactorizar, cambiar y rediseñar.

- DIP → DEPENDENCY INVERSION PRINCIPLE:
    - Los módulos de alto nivel (lógica de negocio principal o funcionalidades más generales) no deben importar nada de los módulos de bajo nivel (implementaciones detalladas de mecanismos individuales). Ambos deben depender de abstracciones (por ejemplo, interfaces).
    - Las abstracciones no deben depender de detalles. Los detalles (implementaciones concretas) deben depender de las abstracciones.

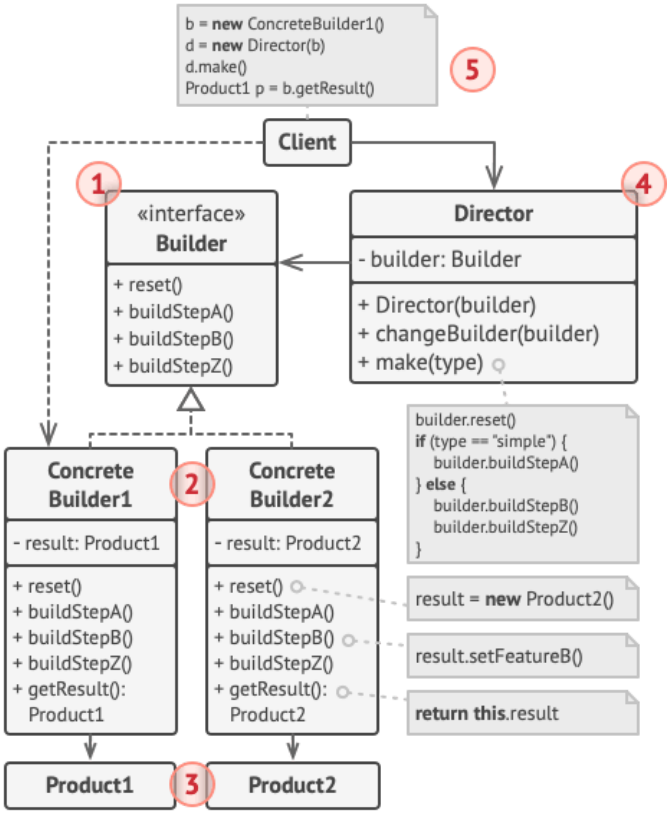
  En muchos casos, pensar en la interacción como un concepto abstracto permite reducir el acoplamiento de los componentes sin introducir patrones de codificación adicionales, permitiendo solo un esquema de interacción más ligero y menos dependiente de la implementación.



DESIGN PATTERNS

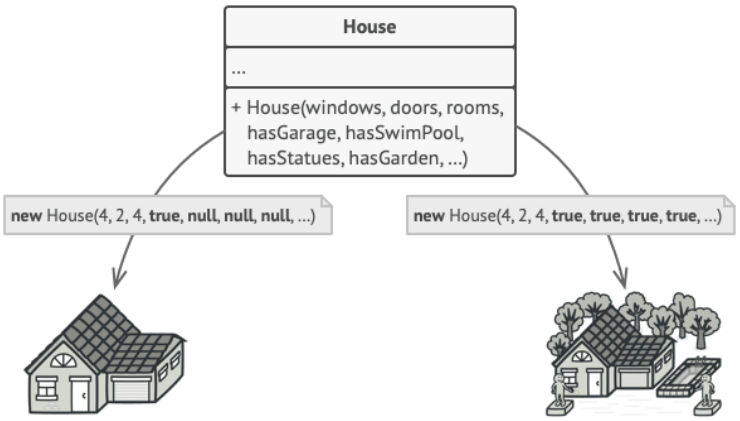
# CREATIONAL

1. BUILDER: Permite construir objetos complejos paso a paso. El patrón permite producir diferentes tipos y representaciones de un objeto utilizando el mismo código de construcción. Algunos de los pasos de construcción pueden requerir diferentes implementaciones cuando se necesita construir varias representaciones del producto. En este caso, se pueden crear varias clases de constructor que implementen el mismo conjunto de pasos de construcción de manera diferente.
    - Director: Se puede ir más allá y extraer una serie de llamadas a los pasos del constructor que se usan para construir un producto en una clase separada llamada director. La clase director define el orden en que se ejecutan los pasos de construcción, mientras que el constructor proporciona la implementación de esos pasos.

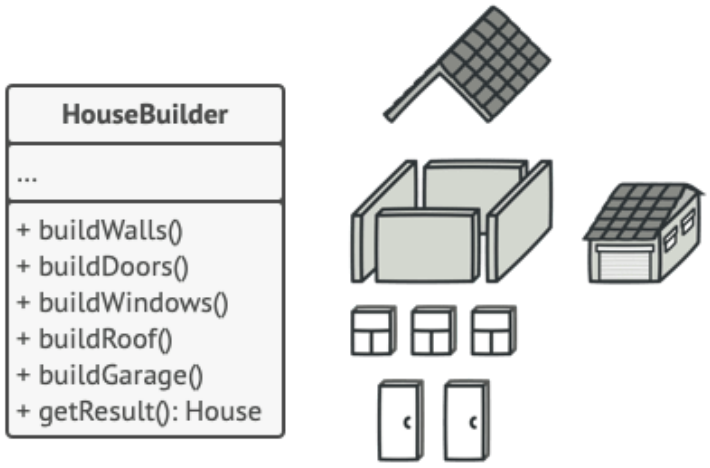



Aplicabilidad: Si tienes un constructor con diez parámetros opcionales, llamarlo puede ser muy inconveniente. Por lo tanto, se sobrecarga el constructor y se crean varias versiones más cortas con menos parámetros. Estos constructores aún se refieren al principal, pasando algunos valores predeterminados a cualquier parámetro omitido. El patrón Builder puede aplicarse cuando la construcción de varias representaciones del producto implica pasos similares que solo difieren en los detalles.
PROS: SRP, step-by-step construction, reuse construction code.

CONS: complexity.
EJEMPLO: Crear un objeto Casa



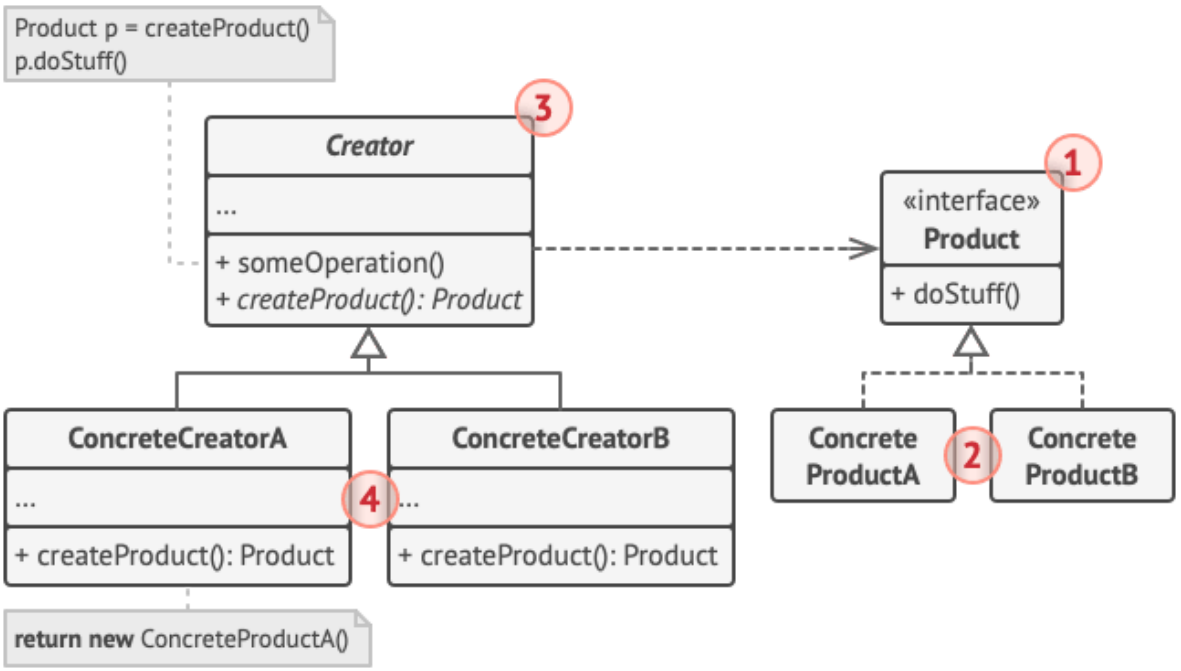

The constructor with lots of parameters has its downside: not all the parameters are needed at all times.



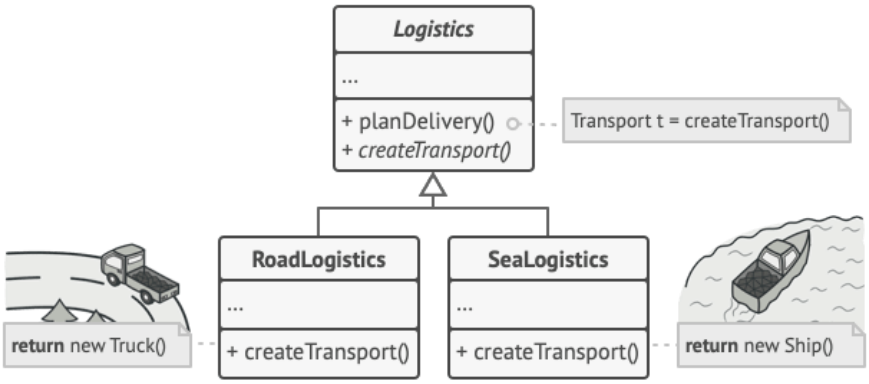

The Builder pattern lets you construct complex objects step by step. The Builder doesn't allow other objects to access the product while it's being built.

2. FACTORY: Proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán. Sugiere reemplazar las llamadas directas de construcción de objetos (usando el operador new) con llamadas a un método de Creator. Se puede sobrescribir el método de fábrica en una subclase y cambiar la clase de productos que se crean mediante el método.
   Aplicabilidad: separa el código de construcción de productos del código que realmente utiliza el producto. Por lo tanto, es más fácil extender el código de construcción de productos independientemente del resto del código. Úsalo cuando quieras proporcionar a los usuarios de tu biblioteca o framework una forma de extender sus componentes internos. Úsalo cuando quieras ahorrar recursos del sistema reutilizando objetos existentes en lugar de reconstruirlos cada vez.
   - Abstract Factory: Puedes declarar el factory method as abstract como abstracto para obligar a todas las subclases a implementar sus propias

versiones del método. Como alternativa, el método de fábrica base puede devolver algún tipo de producto predeterminado.



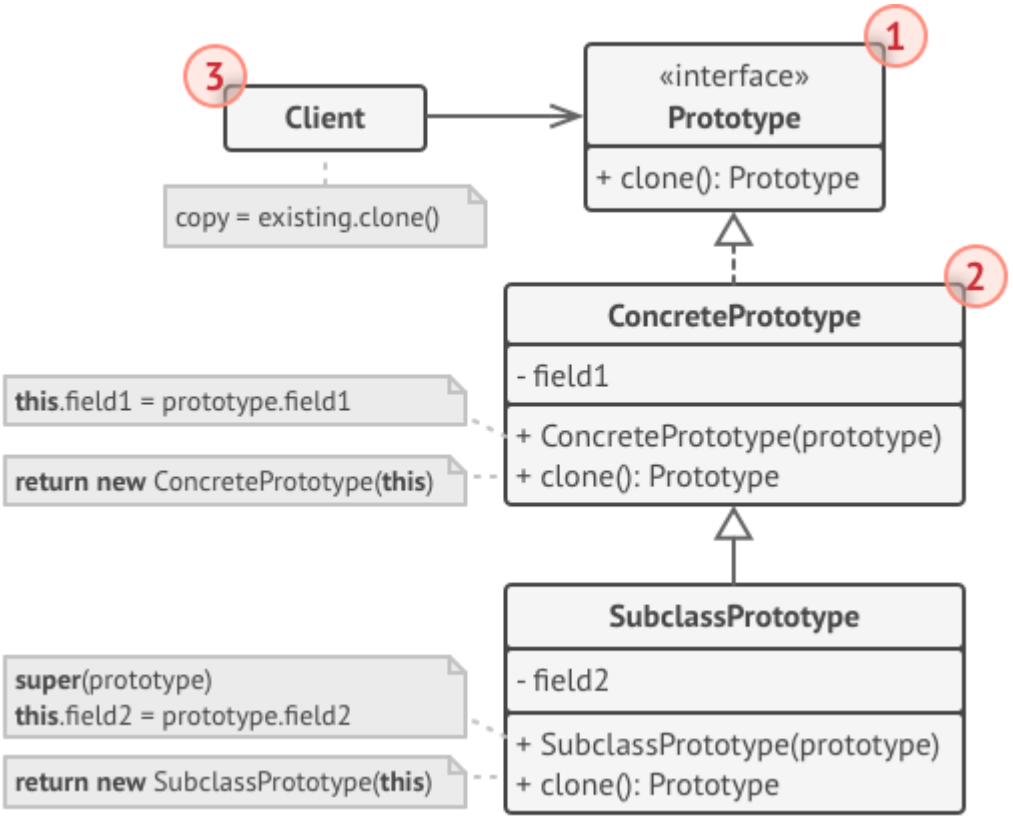

PROS: avoid tight coupling between creator and products, SRP, OCP.
CONS: complexity
EJEMPLO: Empresa logística que comenzó solo con camiones, pero ahora quiere ampliar sus medios de transporte.




limitation: subclasses may return different types of products only if these products have a common base class or interface.

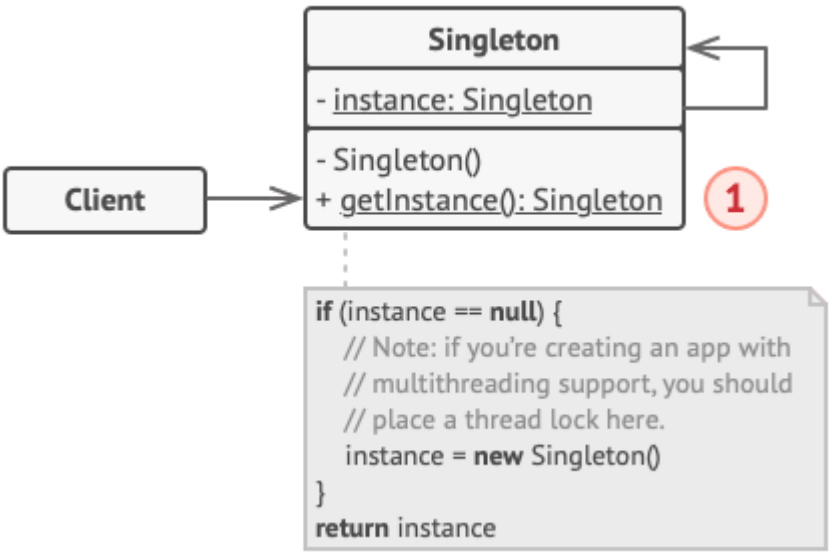
3. PROTOTYPE: Permite copiar objetos existentes sin hacer que tu código dependa de sus clases. El patrón Prototype delega el proceso de clonación a los objetos que se están clonando. El patrón declara una interfaz común para todos los objetos que admiten la clonación. Esta interfaz te permite clonar un objeto sin acoplar tu código a

la clase de ese objeto. Usualmente, dicha interfaz contiene solo un método de clonación.




Aplicabilidad: Usar cuando tu código no deba depender de las clases concretas de los objetos que necesitas copiar. Te permite utilizar un conjunto de objetos pre-built configurados de diversas formas como prototipos. En lugar de instanciar una subclase que coincida con alguna configuración, el cliente simplemente puede buscar un prototipo apropiado y clonarlo.
PROS: clone objects without coupling to their concrete classes, get rid of repeated initialization code in favor of cloning pre-built prototypes.
CONS: cloning complex objects with circular references might be very tricky.

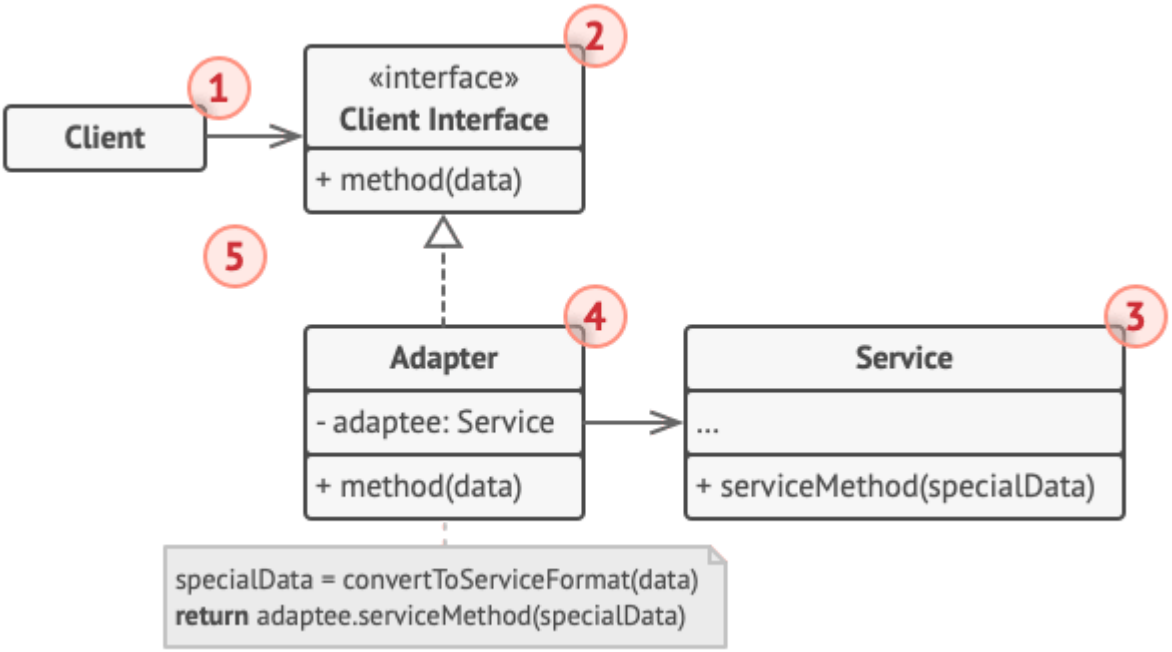
4. SINGLETON: Permite asegurar que una clase tenga solo una instancia (para controlar el acceso a un recurso compartido), mientras proporciona un punto de acceso global a esta instancia. En lugar de crear un nuevo objeto con un constructor, siempre se llama al mismo objeto.
   Convierte al constructor en una clase privada. Crea un static method que actúe como el constructor, y solo este tendrá acceso al constructor, siempre devolverá la misma instancia.




Aplicabilidad: Usar cuando una clase en tu programa deba tener una sola instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por diferentes partes del programa. Usar cuando es necesario un control estricto sobre variables globales.
PROS: ensure that a class has only a single instance, gain global access point to that instance, initialized only the first time it is requested.
CONS: violates SRP, can mask bad design when the components know too much about each other. requires special treatment in multithread, difficult to do unit tests.

# STRUCTURAL

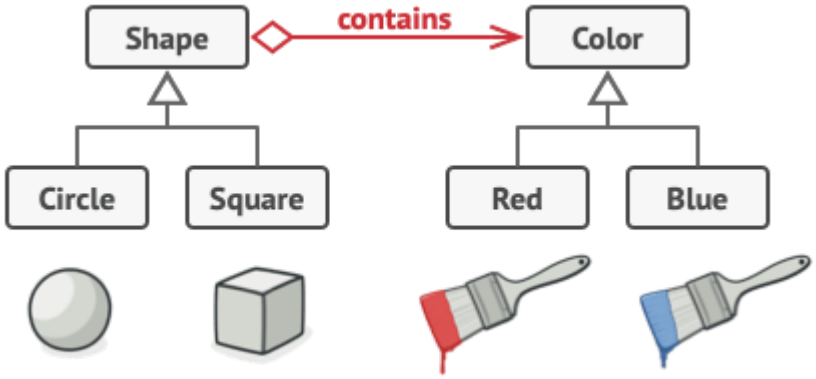
1. ADAPTER: Permite que objetos con interfaces incompatibles colaboren. El adaptador obtiene una interfaz compatible con uno de los objetos existentes. Usando esta interfaz, el objeto existente puede llamar de forma segura a los métodos del adaptador. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que el segundo objeto espera. A veces, es posible crear un adaptador bidireccional que pueda convertir las llamadas en ambas direcciones.

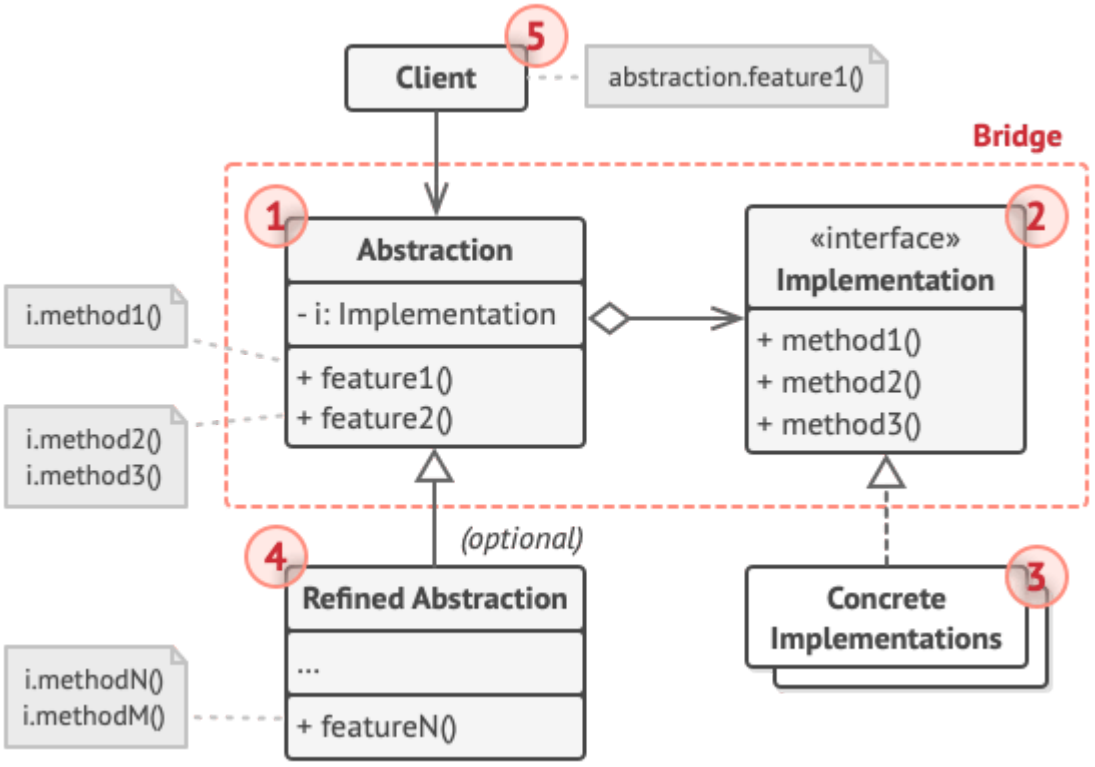



- También se puede utilizar herencia para que el Adapter herede el comportamiento del cliente y del servicio.

Aplicabilidad: Usa la clase Adapter cuando quieras usar alguna clase existente, pero su interfaz no sea compatible con el resto de tu código. Usa el patrón cuando quieras reutilizar varias subclases existentes que carecen de alguna funcionalidad común que no se pueda agregar a la superclase.
PROS: SRP, OCP.
CONS: Complexity, sometimes it's simpler just to change the service class so that it matches the rest of your code.

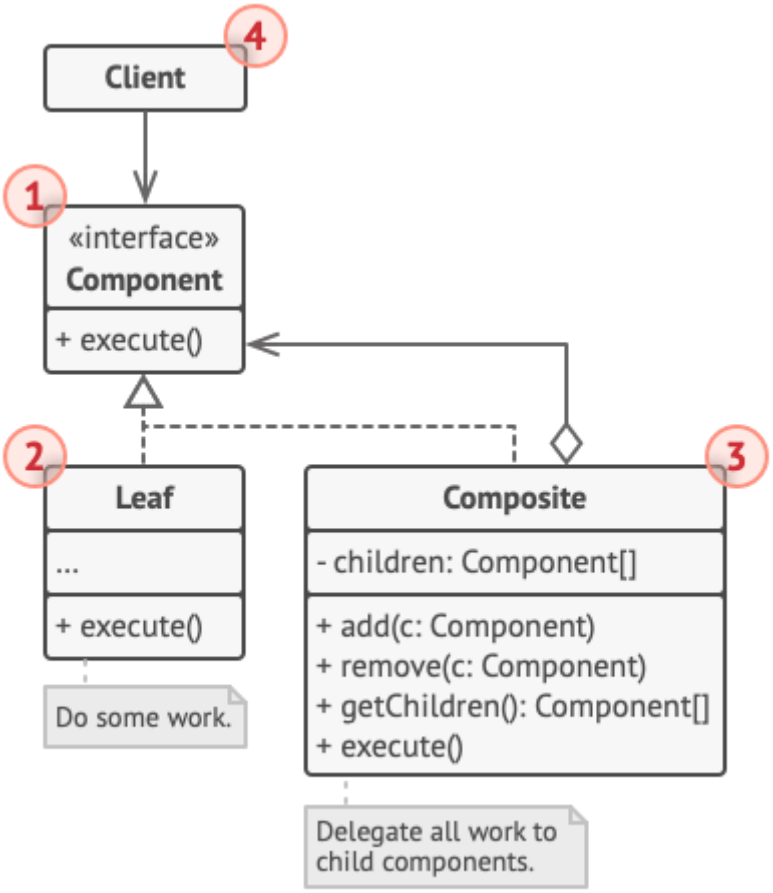
2. BRIDGE: Permite dividir una clase grande o un conjunto de clases estrechamente relacionadas en dos jerarquías separadas: abstracción (interfaz, delega el trabajo) e implementación (realiza el trabajo), que pueden desarrollarse de manera independiente entre sí.
   EJEMPLO: Agregar nuevos tipos de formas y colores a la jerarquía hará que crezca exponencialmente. Solución: cambiar de la herencia a la composición de objetos.



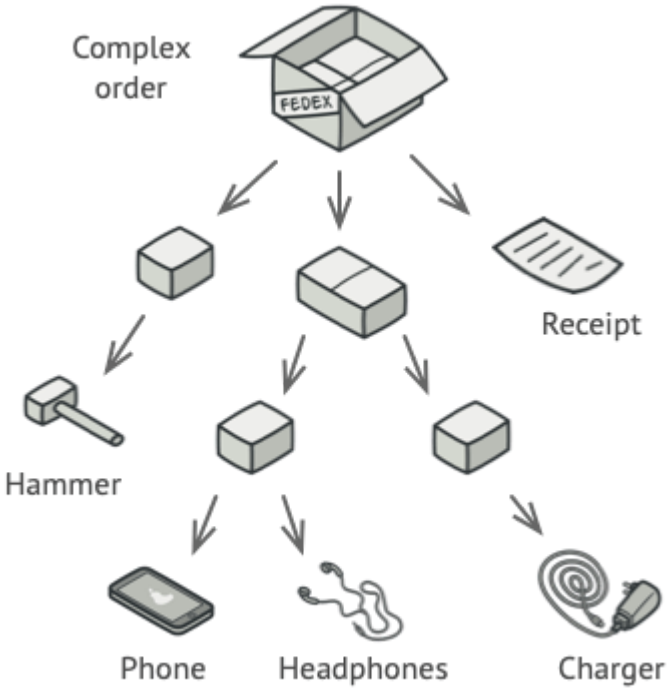



Aplicabilidad: Usar cuando quieras dividir y organizar una clase monolítica que tenga varias variantes de alguna funcionalidad. Usar cuando necesites extender una clase en varias dimensiones ortogonales (independientes). Usar si necesitas poder cambiar implementaciones en tiempo de ejecución.
PROS: create platform-independent classes and apps, high-level abstractions, OCP, SRP.
CONS: Complexity by applying to a highly cohesive class.

3. COMPOSITE: Permite componer objetos en estructuras de árbol y luego trabajar con estas estructuras como si fueran objetos individuales.



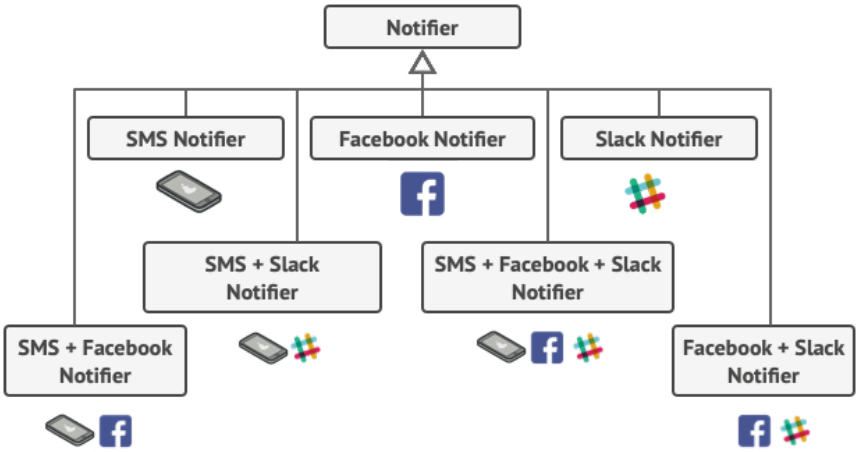

Aplicabilidad: Usar cuando tengas que implementar una estructura de objetos similar a un árbol: dos tipos básicos de elementos que comparten una interfaz común: hojas simples y contenedores complejos. Usa el patrón cuando quieras que el código cliente trate tanto a los elementos simples como a los complejos de manera uniforme.
PROS: work with complex tree structures, OCP.
CONS: it might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.
EJEMPLO: Calcular el precio de una entrega



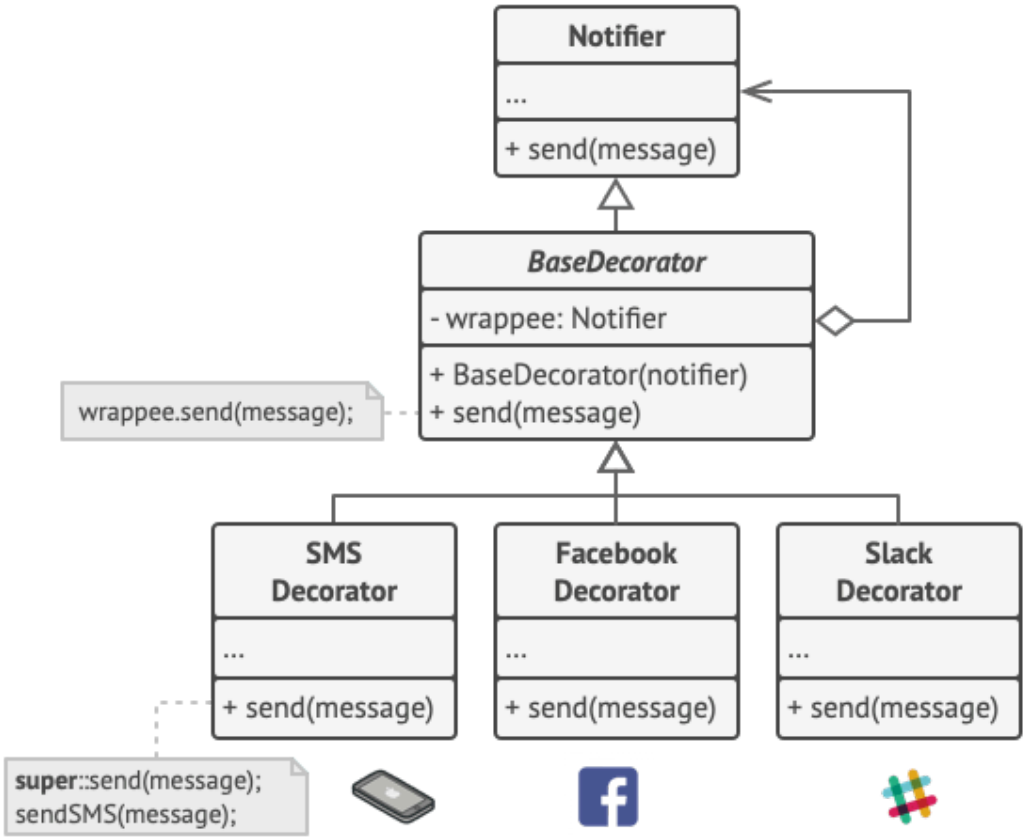

How would this method work? For a product, it'd simply return the product's price. For a box, it'd go over each item the box contains, ask its price and then return a total for this box. If one of these items were a smaller box, that box would also start going over its contents and so on, until the prices of all inner components were calculated. A box could even add some extra cost to the final price, such as packaging cost.

4. DECORATOR: Permite adjuntar nuevos comportamientos a objetos colocando estos objetos dentro de objetos wrapper que contienen los comportamientos.
   A wrapper is an object that can be linked with some target object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives.
   EJEMPLO WRAPPER: Usar ropa. When you're cold, you wrap yourself in a sweater. If you're still cold with a sweater, you can wear a jacket on top. If it's raining, you can put on a raincoat. All of these garments "extend" your basic behavior but aren't part of you, and you can easily take off any piece of clothing whenever you don't need it.

   EJEMPLO: notificar a cliente vía diferentes medios.

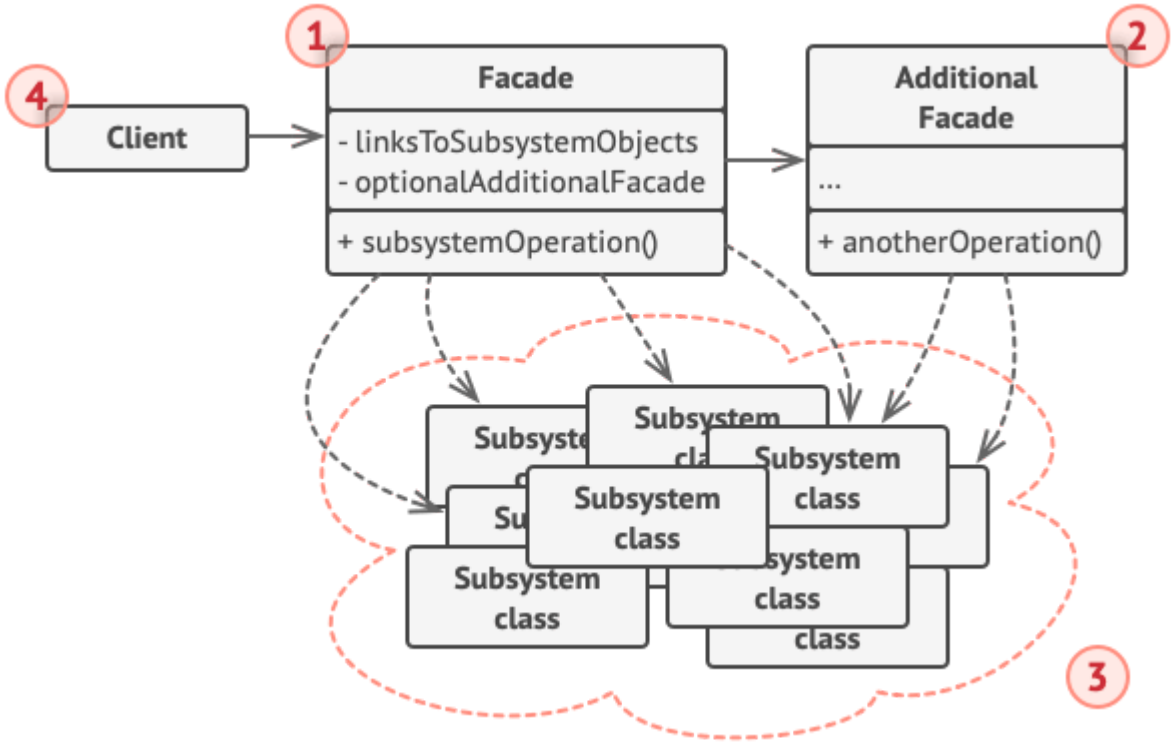



Solución: Convertir los métodos de notificación en Decorators.




Aplicabilidad: Usar cuando necesites poder asignar comportamientos adicionales a objetos en tiempo de ejecución sin romper el código que usa estos objetos. Usa el patrón cuando sea incómodo o no sea posible extender el comportamiento de un objeto mediante herencia.
PROS: extend behavior without making a new subclass. add or remove responsibilities from an object at runtime. combine several behaviors by wrapping an object into multiple decorators. SRP.
CONS: hard to remove a specific wrapper from the wrapper stack. hard to implement so that its behavior doesn't depend on the order in the decorators stack. code layers might look ugly.

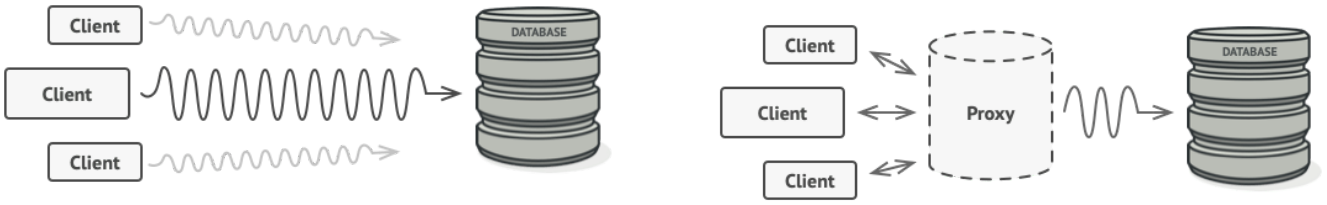
5. FACADE: Proporciona una interfaz simplificada para una biblioteca, un framework o cualquier otro conjunto complejo de clases. Podría proporcionar funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, incluye solo aquellas características que realmente le importan a los clientes.

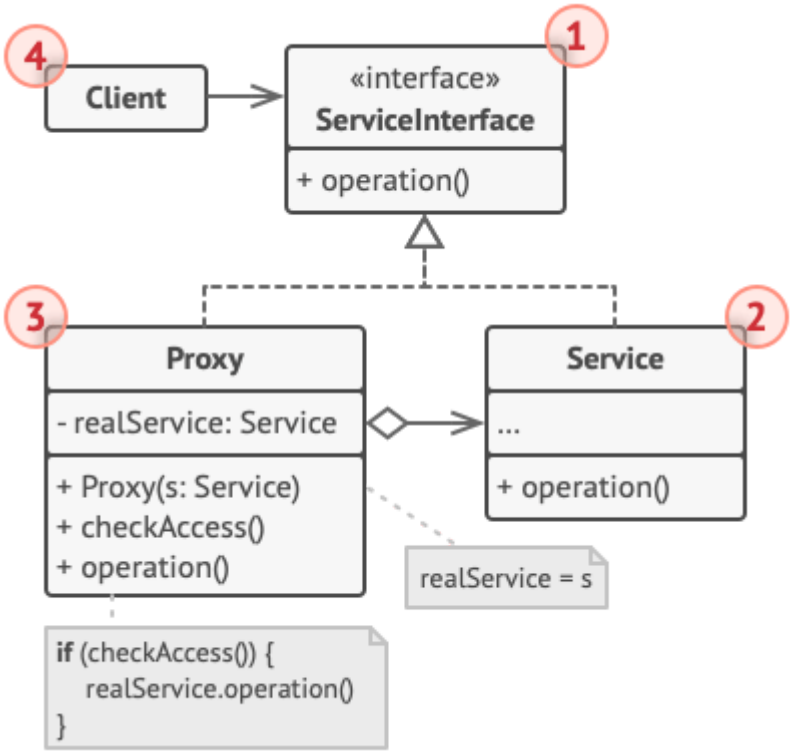



Facade: sabe a dónde dirigir los pedidos de los clientes.
Additional Facade: para evitar que Facade contenga features sin relación. Utilizado por clientes y otras facades.
Aplicabilidad: Usar cuando necesites tener una interfaz limitada pero directa a un subsistema complejo. Crea Facades para definir puntos de entrada a cada nivel de un subsistema. Puedes reducir el acoplamiento entre múltiples subsistemas al requerir que se comuniquen solo a través de Facades.
PROS: isolate code from the complexity of a subsystem.
CONS:  a facade can become a god object coupled to all classes of an app.

6. PROXY: Permite proporcionar un sustituto o un placeholder para otro objeto. Un proxy controla el acceso al objeto original, lo que te permite realizar algo antes o después de que la solicitud llegue al objeto original.
El patrón Proxy sugiere que crees una nueva clase de proxy con la misma interfaz que un objeto de servicio original. Luego, actualiza tu aplicación para que pase el objeto de proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y delega todo el trabajo a él.

EJEMPLO:



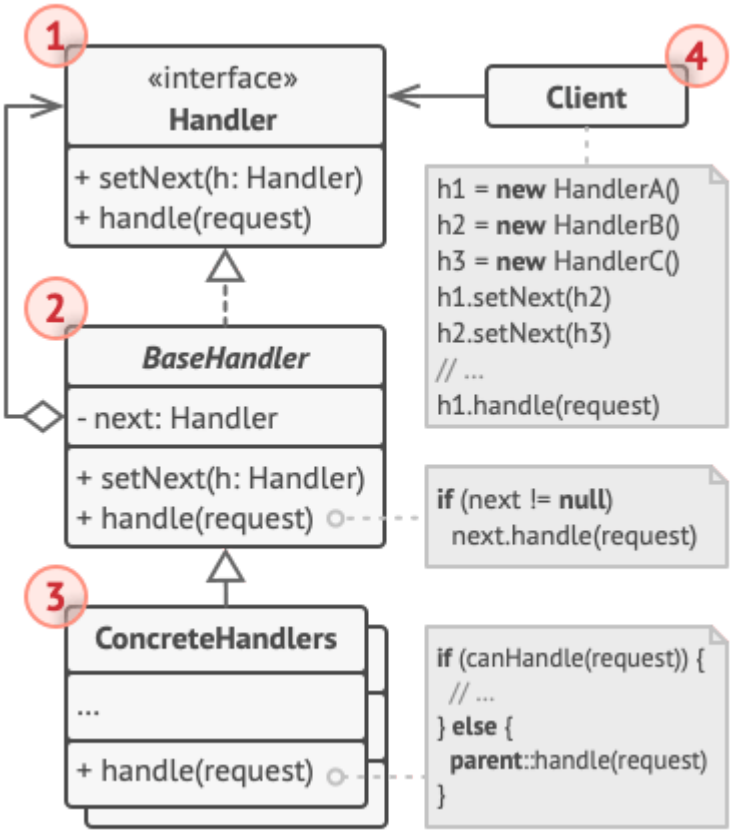

Aplicabilidad:

- Proxy virtual: un objeto de servicio pesado que desperdicia recursos del sistema al estar siempre activo, aunque solo lo necesites de vez en cuando, ahora se inicializa sólo cuando es necesario.
- Proxy de protección: permite que solo clientes específicos utilicen el objeto de servicio; por ejemplo, cuando tus objetos son partes cruciales de un sistema operativo y los clientes son diversas aplicaciones lanzadas (incluidas las maliciosas).
- Proxy remoto: el objeto de servicio está ubicado en un servidor remoto.
- Proxy de registro: mantiene un historial de solicitudes al objeto de servicio.
- Proxy caché: el proxy puede cachear consultas recurrentes.
- Smart reference: el proxy puede liberar recursos cuando el cliente no los usa.

PROS: control services without client knowing, manage lifecycle of the service, the proxy works even if the service object isn't ready or available, OCP.
CONS: complexity. response from the service might get delayed.

# BEHAVIORAL

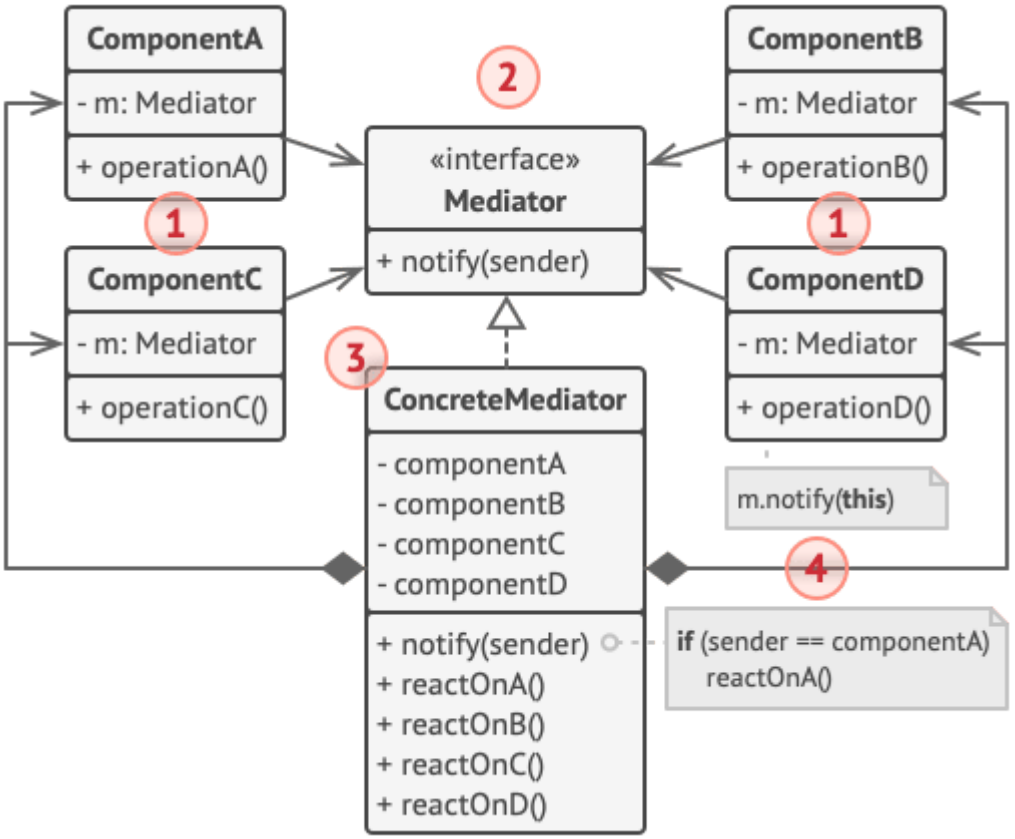
1. CHAIN OF RESPONSIBILITY: Permite pasar solicitudes a lo largo de una cadena de handlers. Al recibir una solicitud, cada manejador decide si procesarla o pasarla al siguiente manejador en la cadena.
   EJEMPLO: en GUI. When a user clicks a button, the event propagates through the chain of GUI elements that starts with the button, goes along its containers (like forms or panels), and ends up with the main application window. The event is processed by the first element in the chain that's capable of handling it.




   Aplicabilidad: Usa este patrón cuando tu programa deba procesar diferentes tipos de solicitudes de diversas maneras, pero los tipos exactos de solicitudes y sus secuencias son desconocidos de antemano. También úsalo cuando sea esencial ejecutar varios manejadores en un orden particular.
   PROS: control the order of request handling, SRP, OCP.
   CONS: some requests may end up unhandled.

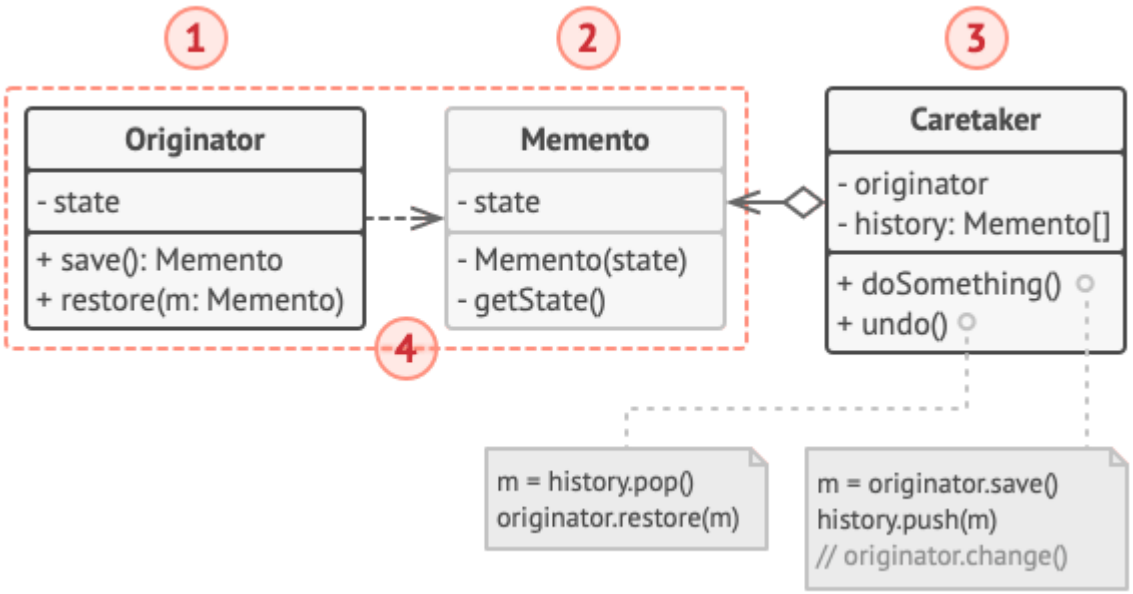
2. MEDIATOR: Reduce las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos y los obliga a colaborar solo a través de un objeto mediador. Como resultado, los componentes dependen únicamente de una clase mediadora en lugar de estar acoplados a docenas de sus colegas.

EJEMPLO: Selecting the “I have a dog” checkbox may reveal a hidden text field for entering the dog’s name. Muy acoplado, separarlo.


El componente no es consciente de la clase real del mediador, por lo que puedes reutilizar el componente en otros programas vinculándolo a un mediador diferente.
Concrete Mediators: encapsulan las relaciones entre varios componentes.
Aplicabilidad: Úsalo cuando sea difícil cambiar algunas de las clases porque están fuertemente acopladas a muchas otras. También úsalo cuando no puedas reutilizar un componente en un programa diferente porque depende demasiado de otros componentes. Usarlo cuando quieras reutilizar comportamiento en varios contextos.
PROS: SRP, OCP, reduce coupling, reuse components.
CONS: over time a mediator can evolve into a God Object.

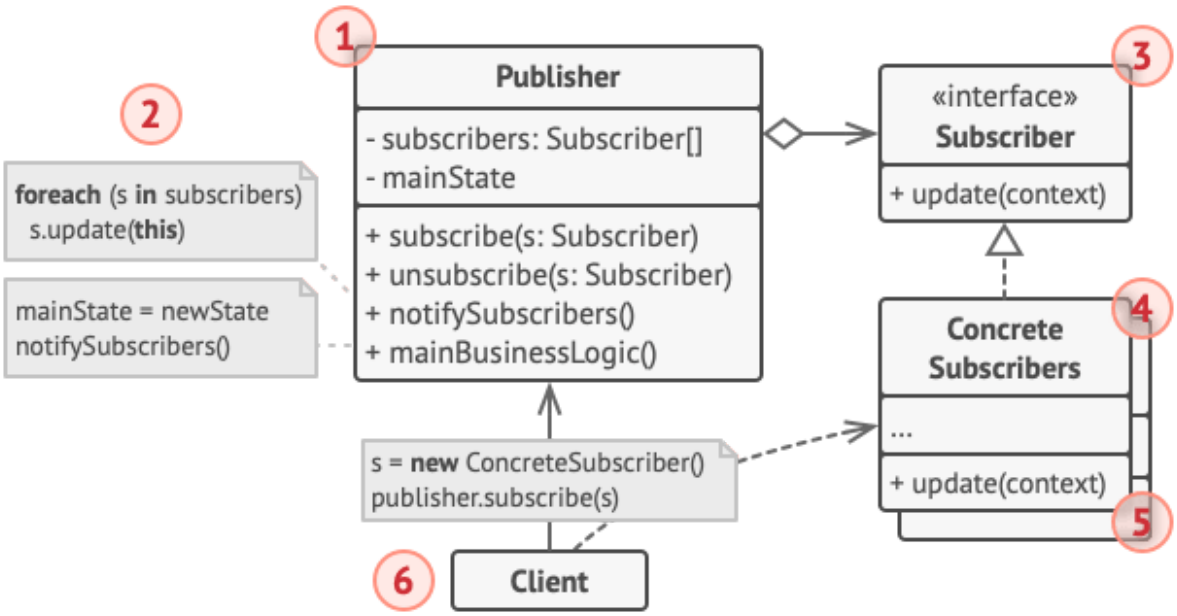
3. MEMENTO: Guarda y restaura el estado anterior de un objeto sin revelar los detalles de su implementación. El patrón sugiere almacenar una copia del estado del objeto en un objeto especial llamado memento. El contenido del memento no es accesible para ningún otro objeto excepto para aquel que lo produjo.




- Caretaker: sabe no solo "cuándo" y "por qué" capturar el estado del originador, sino también cuándo debe restaurarse el estado. Además, puede hacer un seguimiento de la historia del originador almacenando una pila de mementos. Cuando el originador necesita viajar al pasado, el cuidador obtiene el memento más alto de la pila y se lo pasa al método de restauración del originador.
- Es una práctica común hacer que el memento sea inmutable y pasarle los datos solo una vez, a través del constructor.

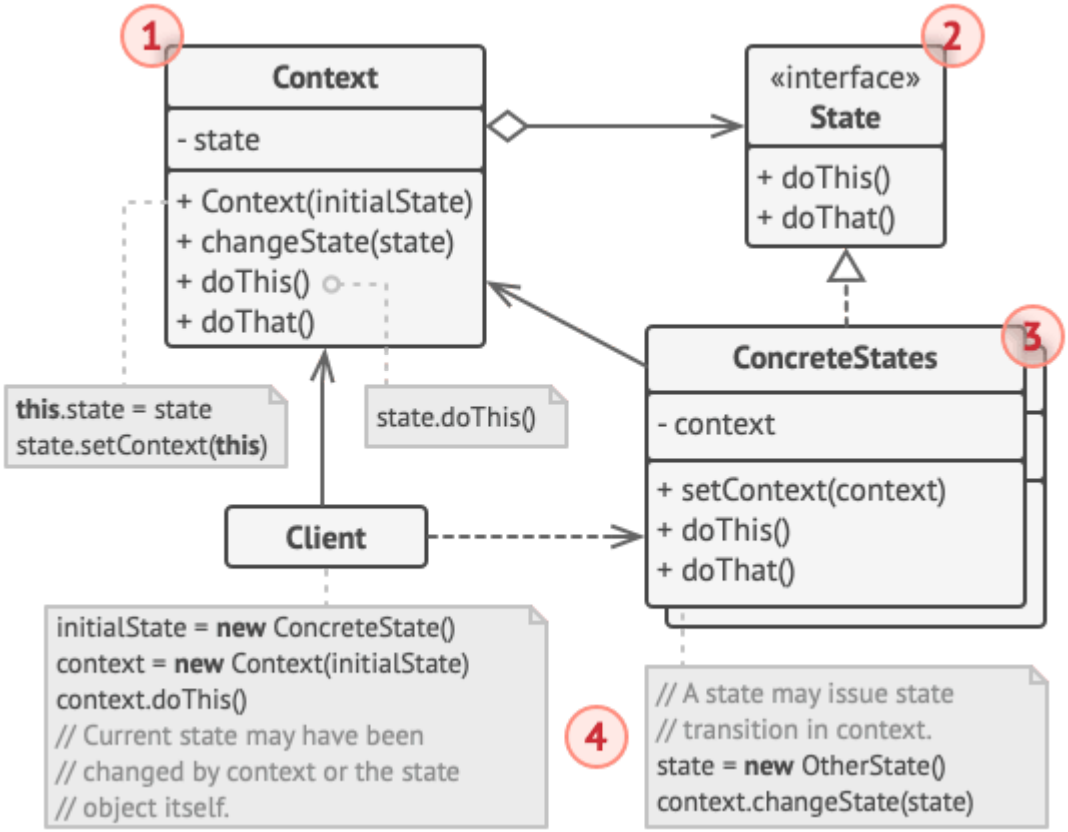
Aplicabilidad: Úsalo cuando desees producir instantáneas del estado del objeto para poder restaurar un estado anterior del objeto. También úsalo cuando el acceso directo a los campos/getters/setters del objeto viola su encapsulamiento.
PROS: produce snapshots without violating its encapsulation. simplify originators code by letting the caretaker maintain history of the originators state.
CONS: consumes a lot of RAM, caretaker tracks the originators lifecycle, most dynamic programming languages can't guarantee that the state within the memento stays untouched.

4. OBSERVER: Permite definir un mecanismo de suscripción para notificar a múltiples objetos sobre eventos que ocurren en el objeto que están observando. El objeto que tiene algún estado interesante a menudo se llama Subject o Publisher. Todos los demás objetos que desean rastrear cambios en el estado del Publisher se llaman Subscribers.
   Este mecanismo consta de 1) un campo de array para almacenar una lista de referencias a objetos subscribers y 2) varios métodos públicos que permiten agregar subscribers y eliminarlos de esa lista.



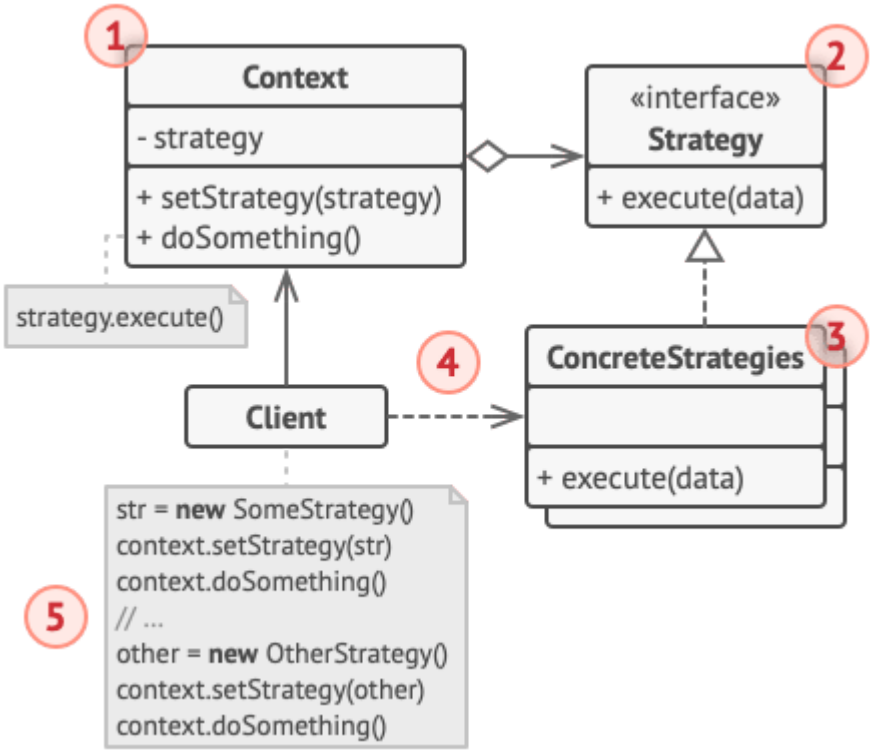

Aplicabilidad: Úsalo cuando los cambios en el estado de un objeto puedan requerir cambios en otros objetos, y el conjunto real de objetos es desconocido de antemano o cambia dinámicamente.
PROS: OCP, establish relations between objects at runtime.
CONS: subscribers are notified in random order.

5. STATE: Permite que un objeto altere su comportamiento cuando su estado interno cambia. Crear nuevas clases para todos los estados posibles de un objeto y extraer todos los comportamientos específicos del estado en estas clases.
   En lugar de implementar todos los comportamientos por sí mismo, el objeto original, llamado Context, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.
   Para cambiar el contexto a otro estado, se reemplaza el objeto de estado activo con otro objeto que representa ese nuevo estado. Esto es posible sólo si todas las clases de estado siguen la misma interfaz y el propio contexto trabaja con estos objetos a través de esa interfaz.

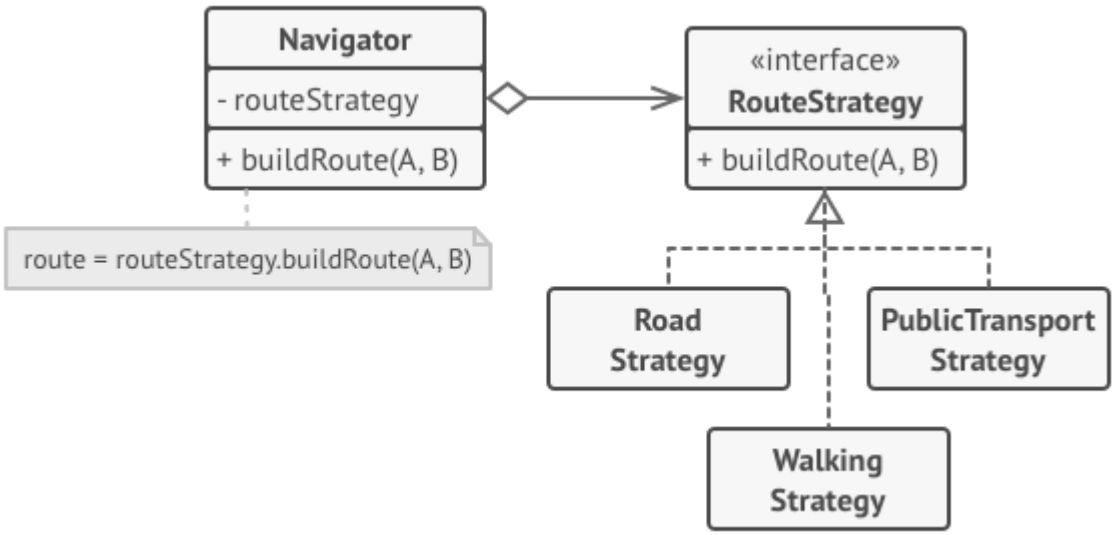



Aplicabilidad: Úsalo cuando tengas un objeto que se comporte de manera diferente según su estado actual, el número de estados sea enorme y el código específico del estado cambie frecuentemente. Usar cuando una clase tiene condicionales masivos que alteran el comportamiento de la clase. Usar cuando hay código duplicado en estados similares.
PROS: SRP, OCP, simplify the code of the context by eliminating state machines.
CONS: Applying the pattern can be overkill if a state machine has only a few states or rarely changes.

6. STRATEGY: Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer que sus objetos sean intercambiables.
   El patrón Strategy sugiere que tomes una clase que hace algo específico de muchas maneras diferentes y extraigas todos estos algoritmos en clases separadas llamadas estrategias.



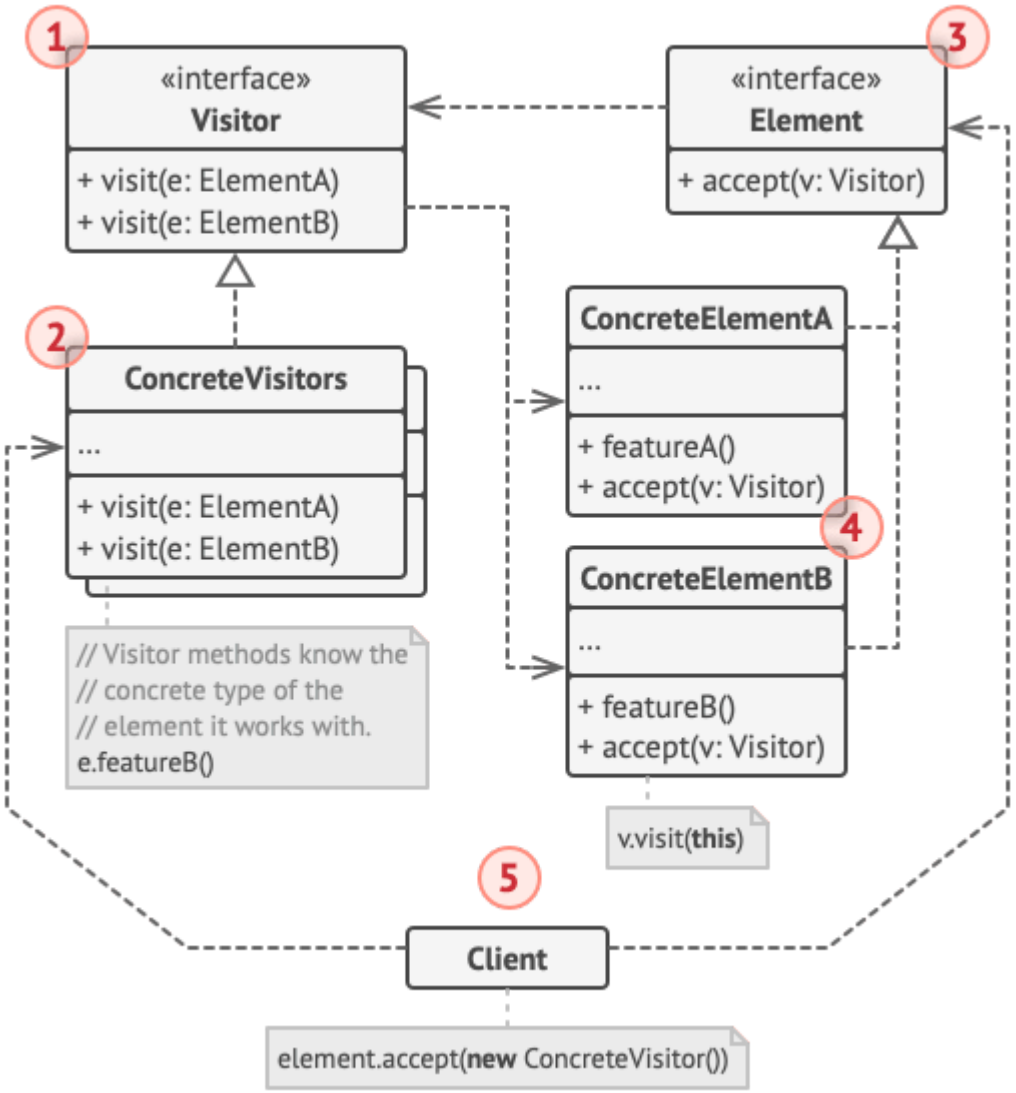

Aplicabilidad: Úsalo cuando desees usar diferentes variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante la ejecución. También úsalo si tienes muchas clases similares que solo difieren en la forma en que ejecutan un comportamiento. Usar para separar la lógica del negocio de la implementación de los algoritmos. Usar cuando la clase tenga condicionales masivos que se diferencian en variantes del mismo algoritmo.
EJEMPLO:




PROS: swap algorithms used inside an object at runtime, isolate implementation details of an algorithm from the code, replace inheritance with composition, OCP.
CONS: algorithms that rarely change it is not necessary to overcomplicate, clients must be aware of the differences.

Alternative: a lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions.

7. VISITOR: permite separar los algoritmos de los objetos sobre los que operan. El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada Visitor, en lugar de intentar integrarlo en las clases existentes. El objeto original que debía realizar el comportamiento ahora se pasa a uno de los métodos del Visitor como argumento, proporcionando al método acceso a todos los datos necesarios contenidos en el objeto.
   En lugar de que el cliente seleccione el método adecuado, se propone que los propios objetos (sobre los que el Visitor operará) se encarguen de esta tarea. Como cada objeto conoce su propia clase, puede determinar de manera más directa qué método del Visitor debe ser ejecutado.




Aplicabilidad: Utiliza el patrón Visitor cuando tienes una estructura de objetos compleja, como un árbol de objetos, y necesitas realizar operaciones en todos sus elementos. El patrón permite que cada elemento "acepte" un visitante, que luego ejecuta la operación específica para ese tipo de elemento. Usa el patrón Visitor para

separar comportamientos adicionales o auxiliares de la lógica principal de negocio de tus clases. Evita la sobrecarga de clases con métodos innecesarios. Al extraer estos comportamientos en una clase visitante, puedes implementar solo los métodos que son relevantes para ciertas clases, manteniendo otras clases limpias y simples.
PROS: OCP, SRP, accumulate information in visitor objects while traverse complex object structure.
CONS: update every visitor each time a class gets added to or removed from the element hierarchy. Visitors might lack the necessary access to the private fields and methods of the elements that they're supposed to work with.
EJEMPLO: Imagina que estás diseñando un sistema de una biblioteca donde tienes diferentes tipos de elementos: libros, revistas y periódicos. Cada uno de estos elementos tiene su propio conjunto de información y comportamientos específicos. Ahora, supongamos que deseas realizar diferentes operaciones en estos elementos, como calcular el costo de cada uno, mostrar detalles específicos o generar informes para cada tipo de elemento. En lugar de agregar métodos específicos para cada operación en cada clase de elemento (lo que podría llevar a una clase con muchos métodos), puedes usar el patrón Visitor para externalizar estas operaciones en clases separadas.

---

ANTIPATRONES
Los antipatrones son formas literarias que describen soluciones comúnmente utilizadas para un problema que generan consecuencias decididamente negativas. Proporcionan experiencias del mundo real para reconocer problemas recurrentes en la industria del software y ofrecen un remedio detallado para los predicamentos más comunes.

Un patrón de intervención clásico se llama "Pelar la Cebolla". Es una técnica de entrevista que implica tres preguntas (en esta secuencia):

- ¿Cuál es el problema?
- ¿Qué están haciendo otras personas para contribuir a este problema?
- ¿Qué estás haciendo tú para contribuir a este problema?

Esta técnica se utiliza para recopilar información en una organización. Responder estas preguntas aumenta la conciencia de las responsabilidades individuales.

Los patrones y antipatrones documentan algunas de las soluciones alternativas conocidas. En respuesta a los estímulos, solo hay dos tipos de respuestas: improvisadas y fijas. Una respuesta improvisada implica inventar algo en el momento. No es un patrón. Una respuesta fija es algo aprendido o practicado. Es un patrón o antipatrón. Si la respuesta conlleva consecuencias negativas, es un antipatrón. Si implica resultados positivos, es un patrón. Un patrón puede convertirse en un antipatrón si se aplica en el contexto incorrecto. Los patrones de diseño comienzan con una solución recurrente, mientras que los antipatrones comienzan con un problema recurrente.

#A procedural design separates process from data, whereas an object−oriented design merges process and data models, along with partitions.

1) The Blob: Procedural−style design. Lleva a que una clase monopolice el procesamiento, mientras que otras clases encapsulan principalmente datos. El problema clave aquí es que la mayoría de las responsabilidades se asignan a una sola clase. Puede ser el resultado de una asignación inapropiada de requisitos, una colección dispar de atributos y operaciones no relacionadas encapsuladas en una sola clase. La falta general de cohesión de los atributos y operaciones es típica del Blob. Tiene ausencia de diseño orientado a objetos (o de cualquier arquitectura en general). Ocurre cuando los programadores agregan funcionalidad en lugar de nuevas clases.
   Excepción: El antipatrón del Bloque es aceptable cuando se envuelven sistemas heredados. No se requiere particionamiento de software, solo una capa final de código para hacer que el sistema heredado sea más accesible.
   Solución: La solución incluye refactorizar el diseño para distribuir las responsabilidades de manera más uniforme y aislar el efecto de los cambios, eliminar todas las asociaciones indirectas "acopladas lejanamente" o redundantes.
   Alternativa: En lugar de una refactorización de abajo hacia arriba de toda la jerarquía de clases, puede ser posible reducir la clase Bloque de un controlador a una clase coordinadora. La clase Bloque original gestiona la funcionalidad del sistema; las clases de datos se extienden con parte de su propio procesamiento.

2) Lava Flow: Ocurre cuando el código muerto y la información de diseño olvidada se acumulan en un diseño en constante cambio, similar a un flujo de lava con grumos de material endurecido. Es común en sistemas que comenzaron como proyectos de investigación pero terminaron en producción debido a la falta de prácticas de diseño sólidas y documentación. Esto dificulta el análisis, verificación y prueba del sistema, impacta negativamente en el rendimiento y dificulta la modularización y reutilización del código.
   Excepción:en prototipos desechables de pequeña escala en entornos de investigación y desarrollo, donde se prioriza la velocidad y el proyecto no necesita ser sustentable.
   Solución: La solución refactorizada incluye un proceso de gestión de configuración que elimina el código muerto y evoluciona o refactoriza el diseño hacia una mayor calidad. A medida que se elimina el código muerto sospechoso, se introducen errores. Cuando esto sucede, resiste la tentación de arreglar inmediatamente los síntomas sin comprender completamente la causa del error. Para evitar Lava Flow, es importante establecer interfaces de software a nivel de sistema que sean estables, bien definidas y claramente documentadas.

3) Golden Hammer: Se refiere al uso obsesivo de una tecnología o concepto familiar en una amplia variedad de problemas de software. Cada nuevo producto o esfuerzo de desarrollo se percibe como algo que se debe resolver mejor con el Golden Hammer. En muchos casos, no es adecuado para el problema, pero se dedica un esfuerzo mínimo a explorar soluciones alternativas. Se utilizan las mismas herramientas y productos para una amplia variedad de productos conceptualmente diversos. Esto ocurre cuando se ha realizado una gran inversión en capacitación y/o adquisición de experiencia en un producto o tecnología, y cuando varios éxitos han utilizado un enfoque particular.

Excepción: Si el producto que define las restricciones arquitectónicas es la solución estratégica prevista a largo plazo; por ejemplo, el uso de una base de datos Oracle para almacenamiento persistente y procedimientos almacenados encapsulados para acceso seguro a los datos. Si el producto forma parte de una suite de proveedores que cubre la mayoría de las necesidades de software.
Solución: Implica ampliar el conocimiento de los desarrolladores a través de la educación, la formación y los grupos de estudio de libros para exponer a los desarrolladores a tecnologías y enfoques alternativos. La gestión debe invertir activamente en el desarrollo profesional de los desarrolladores de software, así como recompensar a aquellos que tomen la iniciativa en mejorar su propio trabajo.

4) Spaghetti Code: El Código Espagueti es una estructura de software ad hoc que dificulta su extensión y optimización. A menudo se encuentra en lenguajes no orientados a objetos y se manifiesta como un sistema sin una estructura clara, lo que dificulta su mantenimiento y comprensión con el tiempo. Se caracteriza por tener pocos objetos con métodos extensos y predecibles, lo que dificulta la interacción dinámica entre ellos. Las causas típicas incluyen la falta de experiencia en diseño orientado a objetos, la falta de mentoría y revisiones de código, y la falta de diseño previo a la implementación.
Excepción: El Código Espagueti es aceptable en ciertos casos donde las interfaces son coherentes y solo la implementación es confusa. Por ejemplo, si se trata de encapsular un fragmento de código no orientado a objetos. Si la vida útil del componente es corta y está aislado limpiamente del resto del sistema, entonces cierta cantidad de código deficiente puede ser tolerable.
Solución: La refactorización frecuente del código puede mejorar su estructura, apoyar el mantenimiento del software y permitir el desarrollo iterativo. La refactorización del software (o limpieza de código) es una parte esencial del desarrollo de software.

5) Cut−and−Paste Programming: Reutilizar código copiando y pegando lleva a problemas de mantenimiento. Surge de la idea de que modificar el código existente es más fácil que programar desde cero. A menudo ocurre cuando hay poca familiaridad con las nuevas tecnologías. Conlleva altos costos de mantenimiento.
Excepción: Aceptable solo para sacar el código rápidamente, pero resulta en mayor mantenimiento.
Solución: Formas alternativas de reutilización, incluida la reutilización de cajas negras, reducen los problemas de mantenimiento al contar con código fuente común, pruebas y documentación. Es necesario reestructurar el software para eliminar la clonación, centrándose en la reutilización de cajas negras. En caso de un uso extensivo de la Programación Copiar y Pegar, se recomienda refactorizar el código en bibliotecas reutilizables que se centren en la reutilización de funcionalidades de caja negra.

6) Monster Commit: Se refiere al envío de un gran conjunto de cambios en lugar de cambios más pequeños y frecuentes al repositorio de versiones. Esto dificulta la revisión y comprensión de los cambios, puede causar conflictos y dificultades de integración, y hace que sea más difícil identificar errores.

Excepción:  en situaciones de emergencia donde se necesita implementar rápidamente una gran cantidad de cambios críticos.
Solución: adoptar prácticas de desarrollo de software más ágiles y centradas en el control de versiones, que fomenten el desarrollo incremental y la entrega frecuente de cambios pequeños y manejables al repositorio de control de versiones. Además, es importante educar y capacitar a los miembros del equipo sobre la importancia de enviar cambios de manera regular y en lotes más pequeños para facilitar la revisión, la integración y la detección temprana de problemas.