STRATEGY

Este patrón cumple con tres principios básicos de la orientación a objetos.
- Encapsular el concepto que varía
- Favorece la composición sobre la herencia.
- Programar para una interfaz, no para una implementación.

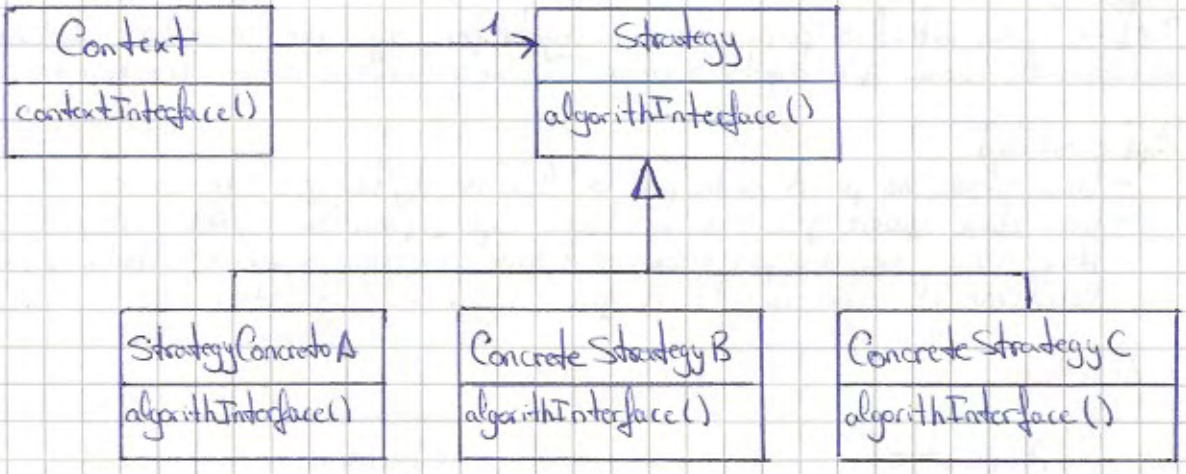
Propósito.

Define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Permite que el algoritmo varíe de forma independiente a los clientes que lo usan.
También conocido como policy

Aplicabilidad.

- Permite configurar una clase con un comportamiento determinado de entre varios.
- Se necesitan distintas variantes de un algoritmo.
- Los distintos comportamientos de una clase aparecen como múltiples sentencias condicionales
  - El patrón Strategy permite mover cada rama de esos condicionales anidados a su propia clase.

Estructura.



Participantes.

- Strategy.
- ConcreteStrategy.
- Context.

Colaboraciones.

- Strategy y Context colaboran para implementar el algoritmo escogido.
  - El contexto puede pasar todos los datos que necesita al llamar a la estrategia concreta.
  - O se puede pasar a sí mismo como referencia para que aquella llame a los métodos que necesite.
- Los clientes colaboran con el contexto.
  - Pueden pasarle el concreteStrategy o no.

Consecuencias
- Define familias de algoritmos relacionados.
- Es una alternativa a la herencia.
  - Hace que el contexto sea más fácil de entender, modificar y mantener.
  - Evita la duplicación de código
  - Evita la explosión de subclases.
  - Se puede cambiar dinámicamente.
- Elimina las múltiples sentencias condicionales.
  - Muchas veces son el indicador de que necesitamos aplicar un patrón (éste u otro, según el caso)
- Elegir entre implementaciones
- Los clientes deben conocer las distintas estrategias.
- Puede complicarse la comunicación entre el contexto y las estrategias.
- Crece el número de objetos

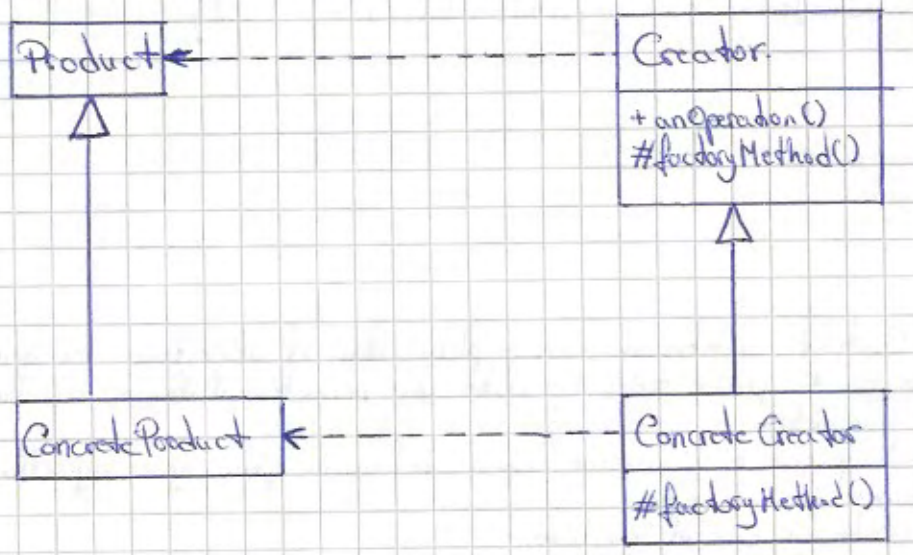
Otros posibles usos
- Validadores de campos de formulario.
- Distintas modalidades de juego.
  - En un juego de póquer, dominó, ajedrez, etc.
  - Niveles de dificultad en el ajedrez o un simulador de caches

FACTORY METHOD

Propósito.
Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan la clase del objeto a crear. También conocido como constructor virtual.

Aplicabilidad.
- Una clase no puede anticipar la clase de objetos que debe crear.
- Una clase quiere que sus subclases especifiquen los objetos a crear.
- Hay clases que delegan responsabilidades en una o varias subclases, y queremos localizar el conocimiento de qué subclase es el delegado.

Estructura.



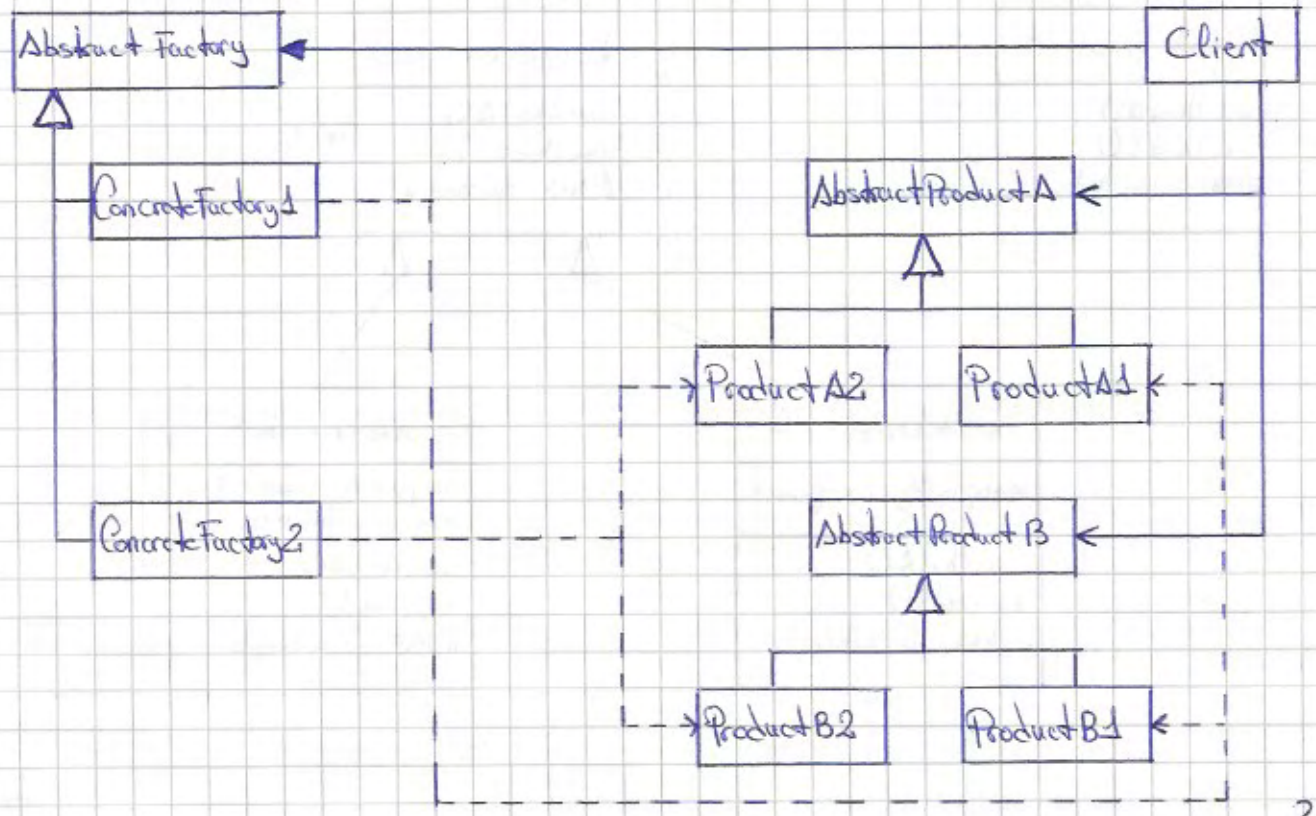
Participantes.
- Product. Una clase no puede anticipar la clase de objetos que debe crear.
- ConcreteProduct. Implementa la interfaz Product.
- Creator.
  - Declara el método de fabricación, que devuelve un objeto de tipo Product.
  - Puede definir una implementación que devuelva el producto concreto predeterminado.
  - Puede llamar a dicho método para crear un objeto producto.
- ConcreteCreator. Redefine el método de fabricación para devolver un objeto ConcreteProduct.

Colaboraciones.
El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

Consecuencias.
- Elimina la necesidad de enlazar clases específicas de la aplicación en el código.
  - Sólo maneja la interfaz Product.
  - Por lo que permite añadir cualquier clase ConcreteProduct definida por el usuario.
- El principal inconveniente es tener que crear una subclase de Creator en los casos en los que ésta no fuera necesaria de no aplicar el patrón.

ABSTRACT FACTORY.
Propósito.
Define una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.

Estructura.

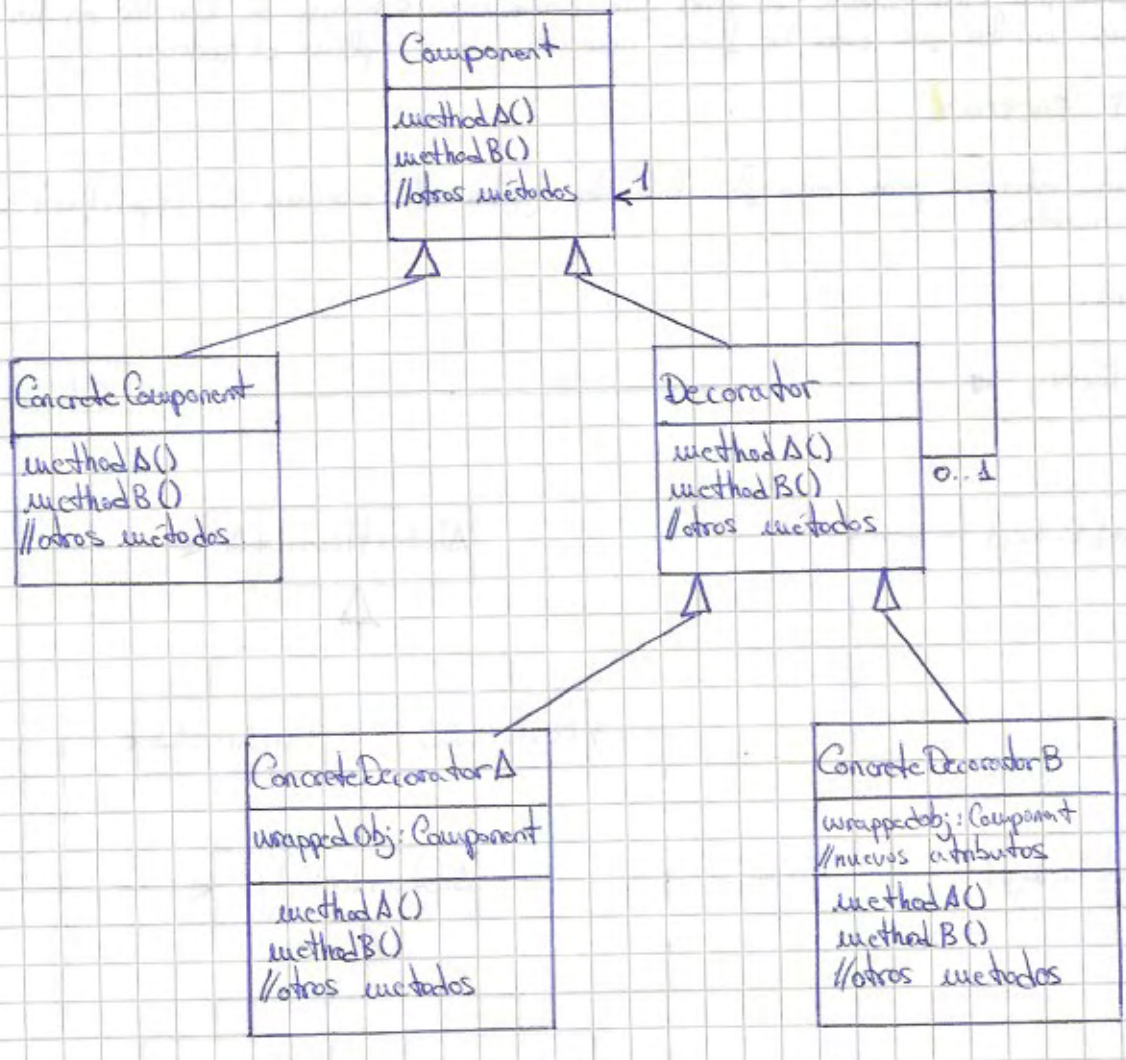


X Nótese como, por el mero hecho de utilizar el patrón, se cumple la restricción de que una barra de desplazamiento de Motif sólo pueda usarse con una ventana de Motif, y...

Consecuencias.
- Aísla las clases concretas. Los clientes manipulan los productos únicamente a través de sus interfaces abstractas, gracias a que las clases de productos concretos están encapsuladas en cada fábrica concreta, no aparecen en el código.
- Permite intercambiar fácilmente familias de productos. Basta con cambiar una única clase, en un único sitio: la fábrica concreta.
- Promueve la consistencia entre los productos. Sólo se pueden usar conjuntamente los objetos de cada familia.
- Dificulta añadir nuevos tipos de productos. Hay que cambiar la interfaz de la fábrica abstracta y por tanto implementar el nuevo método en todas sus subclases.

DECORATOR.
Propósito.
Permite añadir responsabilidades a un objeto dinámicamente. Los decoradores proporcionan una alternativa flexible a la herencia para extender la funcionalidad.
También conocido como wrapper (envoltorio).

Estructura.



Importante.
- Los decoradores tienen el mismo tipo que los objetos que decoran.
  - Podemos usar un decorador en vez del objeto original.
- Se puede usar uno o más decoradores para envolver un objeto
- El decorador añade su propio comportamiento antes o después de delegar al objeto decorado el resto del trabajo.
- Los objetos pueden ser decorados en cualquier momento
  - Podemos decorar objetos dinámicamente en tiempo de ejecución con tantos decoradores como queramos y en cualquier orden.

Aplicabilidad.
- Para añadir responsabilidades a otros objetos dinámicamente y de forma transparente
- Cuando no se puede heredar o no resulta práctico (explosión de subclases para permitir cada combinación posible).

Participantes
- Component. Define la interfaz de los objetos a los que se les puede añadir responsabilidades dinámicamente.
- ConcreteComponent.
- Decorator. Mantiene una interfaz a un objeto Component y tiene su misma interfaz.
- ConcreteDecorator. Añade responsabilidades al componente.

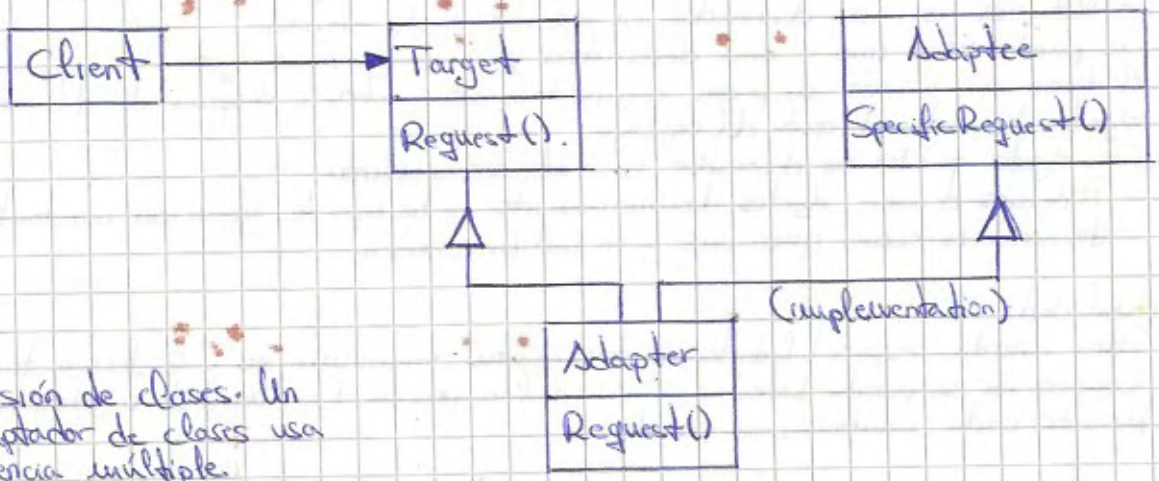
Consecuencias.
- Más flexibilidad que la herencia estática.
- Evita que las clases de arriba de la jerarquía estén repletas de funcionalidades.
  - En vez de definir una clase compleja para tratar de dar cabida a todas ellas, la funcionalidad se logra añadiendo decoradores a una clase simple.
- Un decorador y sus componentes no son idénticos.
  - Desde el punto de vista de la identidad de objetos.
- Muchos objetos pequeños.
  - El sistema puede ser más difícil de aprender y depurar.

ADAPTER.
Propósito.
Convierte la interfaz de una clase en otra que es la que esperan los clientes. Permite que trabajen juntas clases que de otro modo no podrían por tener interfaces incompatibles.

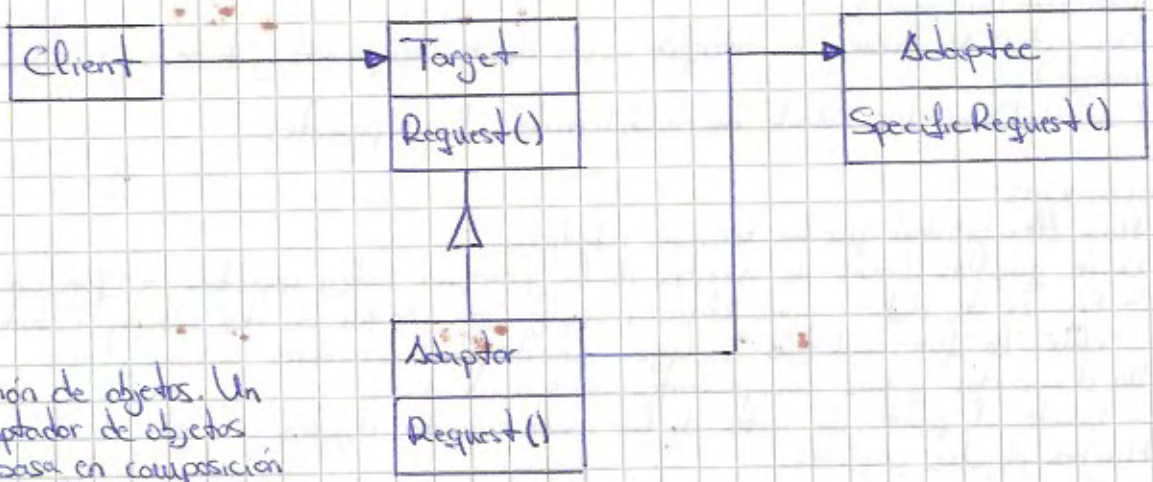
Aplicabilidad.
- Queremos usar una clase existente, y esta no tiene la interfaz (es decir, el tipo) que necesitamos.
- Queremos crear una clase reutilizable que coopere con clases con las que no está relacionada.
  - Que no tendrán, por tanto, interfaces compatibles.

- (Sólo la versión de objetos) Necesitamos usar varias subclases existentes pero sin tener que adaptar su interfaz creando una nueva subclase de cada una.
- Un adaptador de objetos puede adaptar la interfaz de su clase padre.

Estructura.



Versión de clases. Un adaptador de clases usa herencia múltiple.



Versión de objetos. Un adaptador de objetos se basa en composición de objetos.

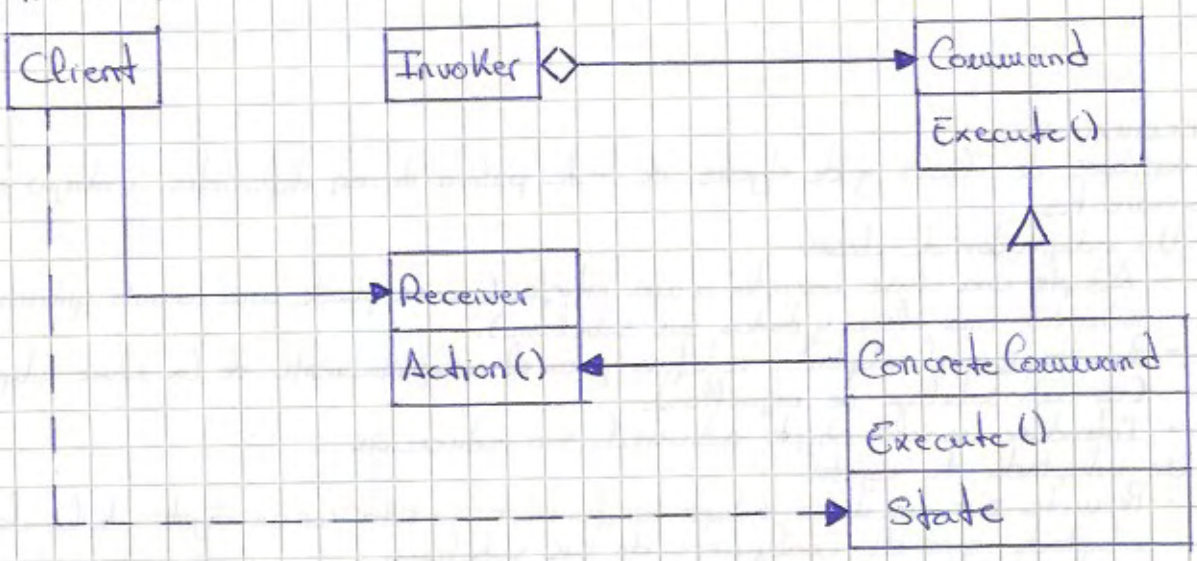
Participantes.
- Target. Define la interfaz específica del dominio (es la que usarán los clientes).
- Client. Colabora con objetos de tipo Target.
- Adaptee. La clase existente cuya interfaz necesita ser adaptada.
- Adapter. Adapta la interfaz de Adaptec a la de Target.

Consecuencias.
Las versiones de clases y de objetos de este patrón tienen diferentes ventajas e inconvenientes.
- Un adaptador de clases
  - Adapta una clase concreta a una interfaz (no se puede usar cuando queremos adaptar una clase y todas sus subclases).
  - Permite que el adaptador redefina para el comportamiento de la clase adaptada (es una subclase de aquéllas).
  - Introduce un solo objeto adicional, sin indirección.
- Un adaptador de objetos
  - Permite que un único adaptador funcione no solo con un objeto de la clase adaptada, sino de cualquiera de sus subclases.
  - No es del tipo del objeto adaptado.

COMMAND.
Propósito.
Encapsula una petición dentro de un objeto, permitiendo parametrizar a los clientes con distintas peticiones, encolarlas, guardarlas en un registro de sucesos o implementar un mecanismo de deshacer/repetir. También conocido como action, transaction.

Aplicabilidad.
- Parametrizar objetos con una determinada acción.
  - Lo que haríamos en un lenguaje de procedimientos con una función callback (como un puntero a función, que será llamada más tarde)
- Que la acción a realizar y el objeto que lanza la petición tengan ciclos de vida distintos
- Permite deshacer/repetir (undo/redo)
  - En este caso, execute deberá guardar el estado para poder revertir los efectos de ejecutar la operación
  - Y hará falta una operación añadida, unexecute.
- Guardar todas las operaciones ejecutadas en un registro (log)
  - Proporcionando un par de operaciones store y load.
  - El sistema podría recuperarse de un error cargando las operaciones guardadas en disco y volviendo a llamar al método execute de cada una de ellas.
- Usar transacciones.

Estructura.



Participantes.
- Command. Define una interfaz para ejecutar una operación.
- ConcreteCommand.
  - Asocia un objeto Receiver con una acción
  - Implementa execute llamando a las operaciones de dicho objeto receptor.
- Client. Crea un objeto ConcreteCommand y establece su receptor.
- Invoker. Ejecuta la acción.
- Receiver.

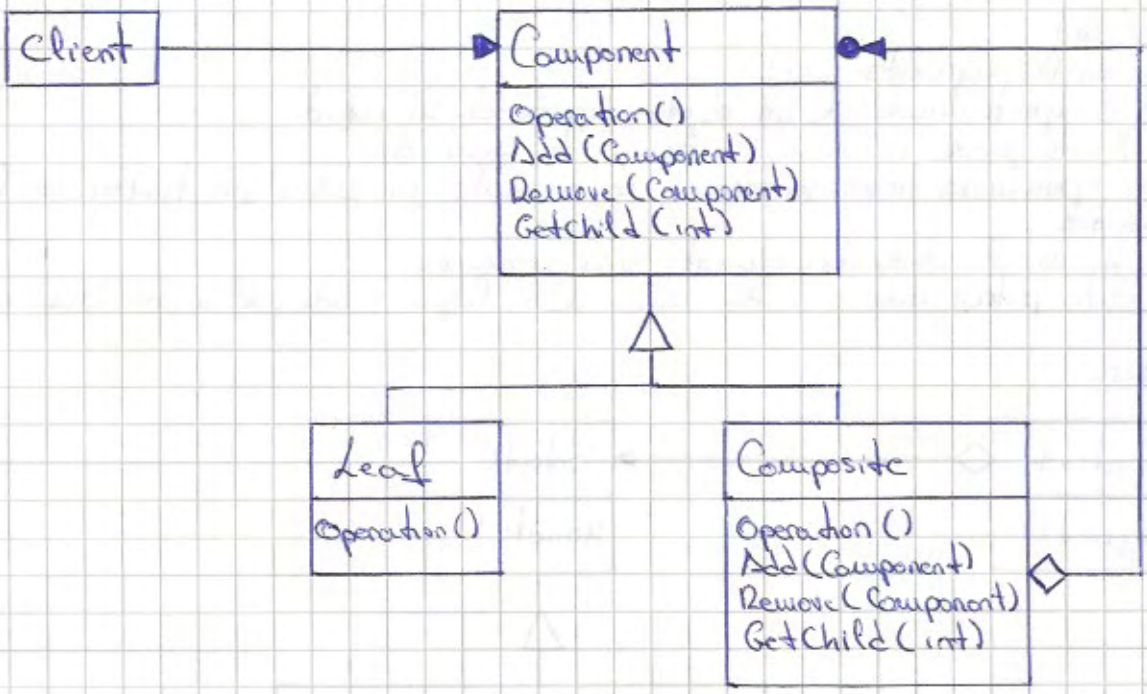
Colaboraciones.
- El cliente crea un objeto ConcreteCommand y especifica su receptor.
- Un objeto Invoker guarda el objeto ConcreteCommand
- Aquel llama a la operación de este último.
  - Quien antes guarda el estado para luego poder deshacer la operación.
- El objeto ConcreteCommand se vale de las operaciones de su receptor para llevar a cabo la acción.

COMPOSITE

Propósito.
Permite componer objetos en estructuras arbóreas para representar jerarquías de todo-parte, de modo que los clientes puedan tratar a los objetos individuales y a los compuestos de manera uniforme.

Aplicabilidad.
- Representar jerarquías de parte-todo.
- Que los clientes traten por igual los objetos individuales y los compuestos.

Estructura.



Participantes.
- Component.
  - Declara la interfaz común.
  - Implementa el comportamiento predeterminado que es común a todas las clases.
  - Declara operaciones para acceder a los hijos
  - (opcional) Define una interfaz para acceder al padre.
- Leaf.
- Composite
  - Almacena sus componentes hijos.
  - Implementa las operaciones relacionadas con los hijos.
- Client.
  - Manipula los objetos de la composición a través de la interfaz de Component.

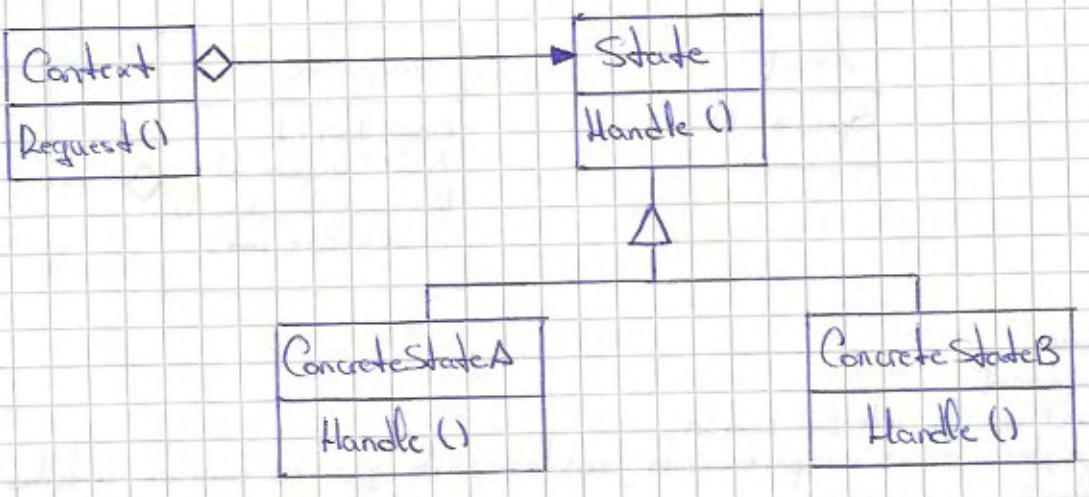
Consecuencias.
- Permite jerarquías de objetos tan complejas como se quiera.
  - Allá donde el cliente espere un objeto primitivo, podrá recibir un compuesto y no se dará cuenta.
- Simplifica el cliente.
  - Al eliminar el código para distinguir entre unos y otros.
- Se pueden añadir nuevos componentes fácilmente.
- Como desventaja, podría hacer el diseño demasiado general.
  - Si queremos restringir los componentes que pueden formar parte de un compuesto determinado.

STATE

Propósito.
Permite a un objeto alterar su comportamiento cuando cambia su estado interno. Parecerá como si el objeto hubiera cambiado su clase. También conocido como estados como objetos.

Aplicabilidad.
Se usa en los siguientes casos.
- El comportamiento de un objeto depende de su estado
  - Y éste puede cambiar en tiempo de ejecución.
- Las operaciones tienen sentencias condicionales unidadas que tratan con los estados.
  - Siendo el estado normalmente una constante.
  - Este patrón mueve cada rama de la lógica condicional a una clase aparte.

Estructura.



Participantes.
- Context.
  - Define la interfaz que interesa a los clientes.
  - Mantiene una referencia a una subclase de estado concreto que representa el estado actual.
- State
  - Define la interfaz para encapsular el comportamiento asociado con el estado del contexto.
- Subclases: ConcreteState.
  - Cada subclase implementa los operaciones anteriores para estado concreto.

Colaboraciones
- El contexto delega las operaciones dependientes del estado al objeto que representa el estado actual.
- El contexto podría pasarse a sí mismo como parámetro
  - Para que el estado acceda al contexto si es necesario.
- Una vez que el contexto es inicializado en un determinado estado, los clientes no necesitan tratar directamente con los estados.
- O bien el contexto o bien los estados concretos deciden cuándo se pasa de uno a otro estado.

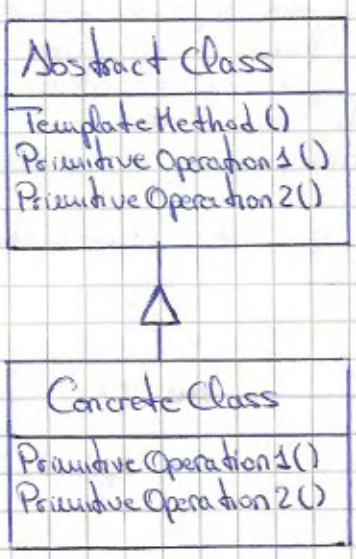
Consecuencias.
- Localiza el comportamiento específico del estado y lo aísla en un objeto.
  - Se pueden añadir nuevos estados y transiciones fácilmente, simplemente definiendo nuevas clases de State.
- Hace explícitas las transiciones entre estados.

TEMPLATE METHOD

Propósito.
Define el esqueleto de un algoritmo en una operación, definiendo algunos pasos hasta las subclases. Permite que éstas redefinan ciertos pasos del algoritmo sin cambiar la estructura del algoritmo en sí.

Aplicabilidad.
- Para implementar las partes de un algoritmo que no cambian y dejar que las subclases implementen aquéllas otras que pueden variar.
- Como motivo de factorizar código, cuando movemos cierto código a una clase base común para evitar código duplicado.
- Para controlar el modo en que las subclases extienden la clase base.
  - Dejando que sea sólo a través de unos métodos de plantilla dados.

Estructura.



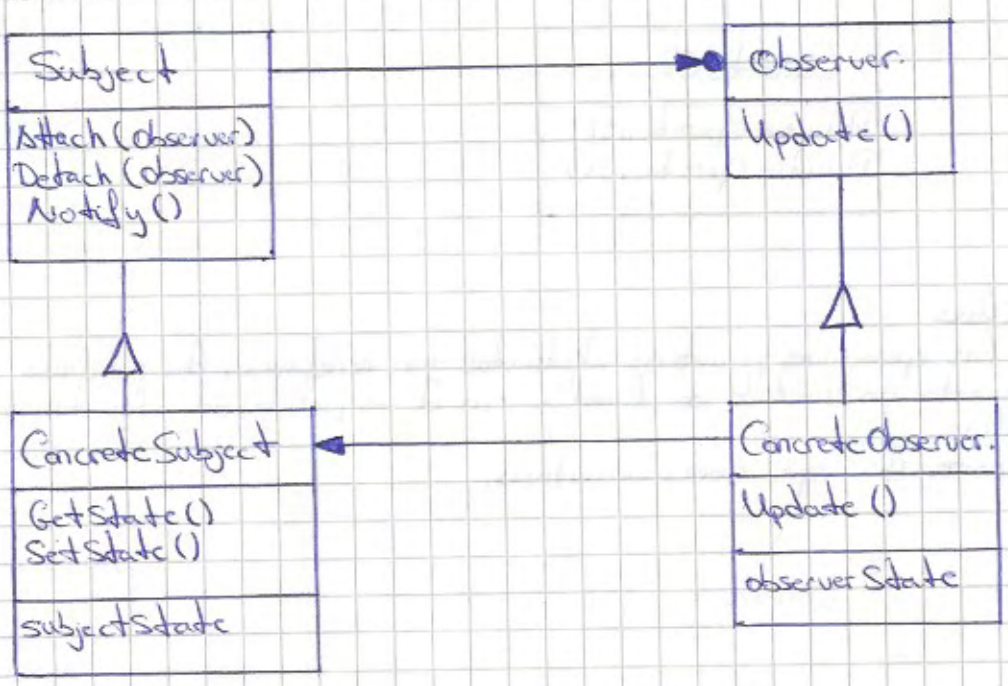
Participantes.
- AbstractClass
  - Define las operaciones primitivas abstractas que redefinirán las subclases.
  - Implementa un método de plantilla con el esqueleto del algoritmo.
- ConcreteClass.
  - Implementa las operaciones primitivas.

Consecuencias.
- Los métodos de plantilla son una técnica fundamental para la reutilización de código.
- Inversión de control. Es la clase padre quien llama a operaciones en los hijos.
- Los métodos de plantilla pueden llamar a los siguientes tipos de operaciones.
  - Operaciones concretas de las subclases o de otras clases.
  - Operaciones concretas en la propia clase base abstracta.
  - Operaciones primitivas (es decir, abstractas).
  - Métodos de fabricación
  - Operación de enganche
    - Proporcionan comportamiento predeterminado que las subclases pueden redefinir si es necesario. Normalmente, la implementación predeterminada no hace nada.

OBSERVER.
Propósito.
Define una dependencia uno-a-muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los demás objetos dependientes se modifican y actualizan automáticamente. También conocido como publicar-suscribir. (publish-subscribe).

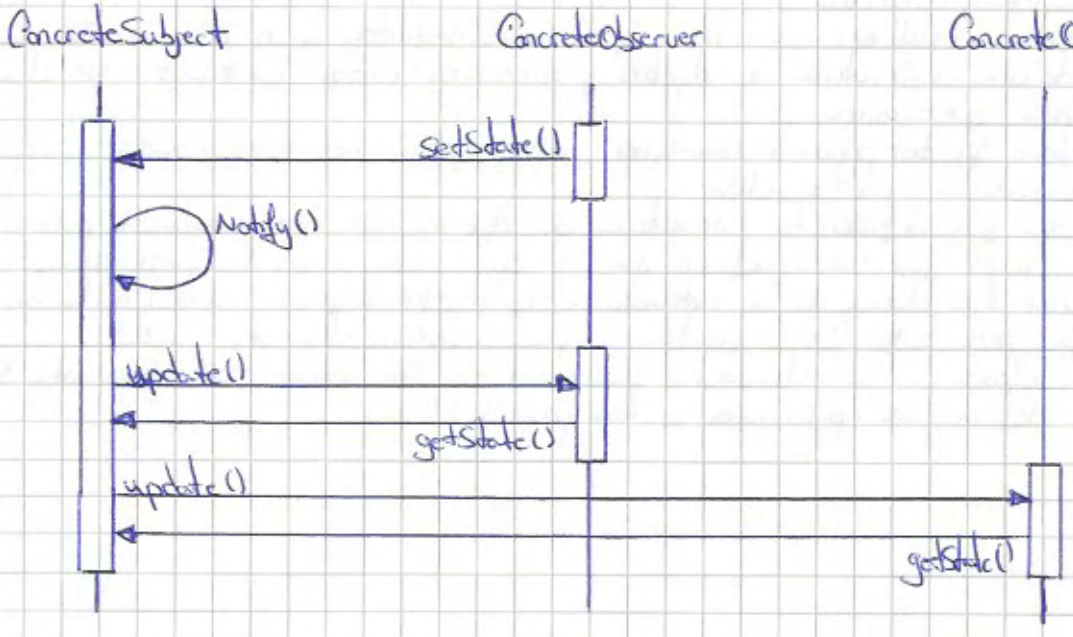
Aplicabilidad.
- Una abstracción tiene dos aspectos, uno de los cuales depende del otro
  - Encapsular estos aspectos en objetos separados permite que los objetos varíen (y puedan ser reutilizados) de forma independiente.
- Un cambio en un objeto requiere que cambien otros.
  - Y no sabemos a priori ni cuáles ni cuántos.
- Un objeto necesita notificar a otros cambios en su estado sin hacer presunciones sobre quiénes son dichos objetos.
  - Es decir, cuando no queremos que estén fuertemente acoplados.

Estructura.



Participantes
- Subject.
  - Conoce a sus observadores.
  - Proporciona una interfaz para que se suscriban los objetos observer (o que se borren).
- Observer.
  - Define una interfaz para actualizar los objetos que deben ser notificados de cambios en el objeto Subject.
- ConcreteSubject.
  - Guarda el estado de interés para los objetos ConcreteObserver.
  - Envía una notificación a sus observadores cuando cambia su estado.
- ConcreteObserver
  - Mantiene una referencia a un objeto ConcreteSubject.
  - Guarda el estado que debería permanecer sincronizado con el objeto observado.
  - Implementa la interfaz Observer para mantener su estado consistente con el objeto observado.

Colaboraciones.
- El objeto observado notifica a sus observadores cada vez que ocurre un cambio. A fin de que el estado de ambos permanezca consistente.
- Después de ser informado de un cambio en el objeto observado, cada observador concreto puede pedirle la información que necesita para reconciliar su estado con el de aquel.

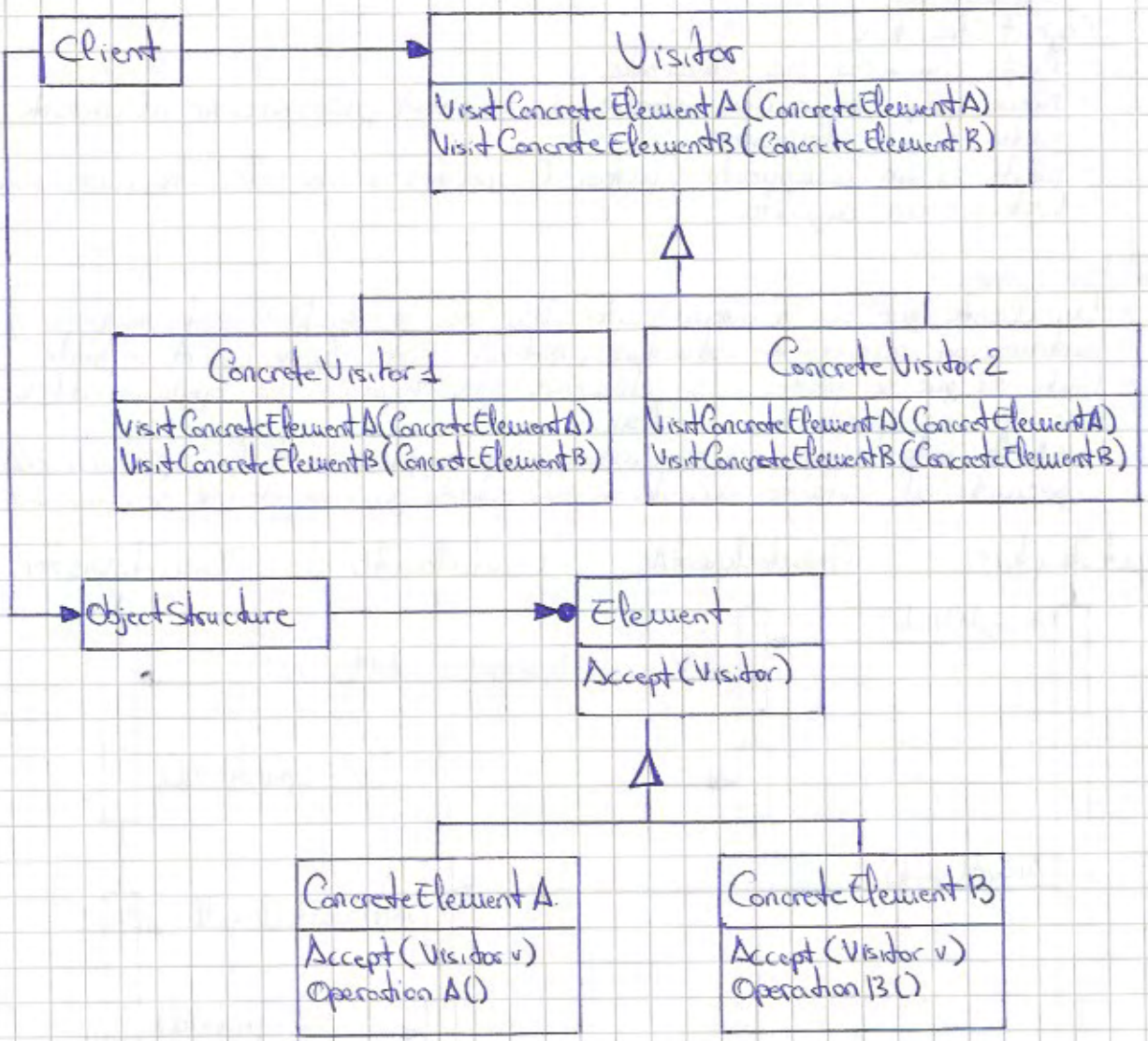


Consecuencias.
- Permite variar objetos observados y observadores independientes.
  - Se pueden reutilizar los objetos observados (Subject) sin sus observadores, y viceversa.
  - Se pueden añadir nuevos observadores sin modificar ninguna de las clases existentes.
- Acoplamiento abstracto entre Subject y Observer.
  - Todo lo que un objeto sabe de sus observadores es que tiene una lista de objetos que satisfacen la interfaz Observer.
    - Con lo que podrían incluso pertenecer a dos capas distintas de la arquitectura de la aplicación.
- No se especifica el receptor de una actualización
  - Se envía a todos los objetos interesados.
- Actualizaciones inesperadas
  - Se podrían producir actualizaciones en cascada muy ineficientes.

VISITOR.
Propósito.
Representa una operación a realizar sobre una estructura de objetos. Permite definir nuevas operaciones sin modificar las clases de los elementos sobre los que opera.

Aplicabilidad
- Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta.
- Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos y queremos evitar 'contaminar' sus clases con dichas operaciones.
  - El patrón Visitor permite mantener juntas operaciones relacionadas definiéndolas en la clase.
- Las clases que definen la estructura de objetos rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura.
  - Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es potencialmente costoso.
  - Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases.

Estructura.

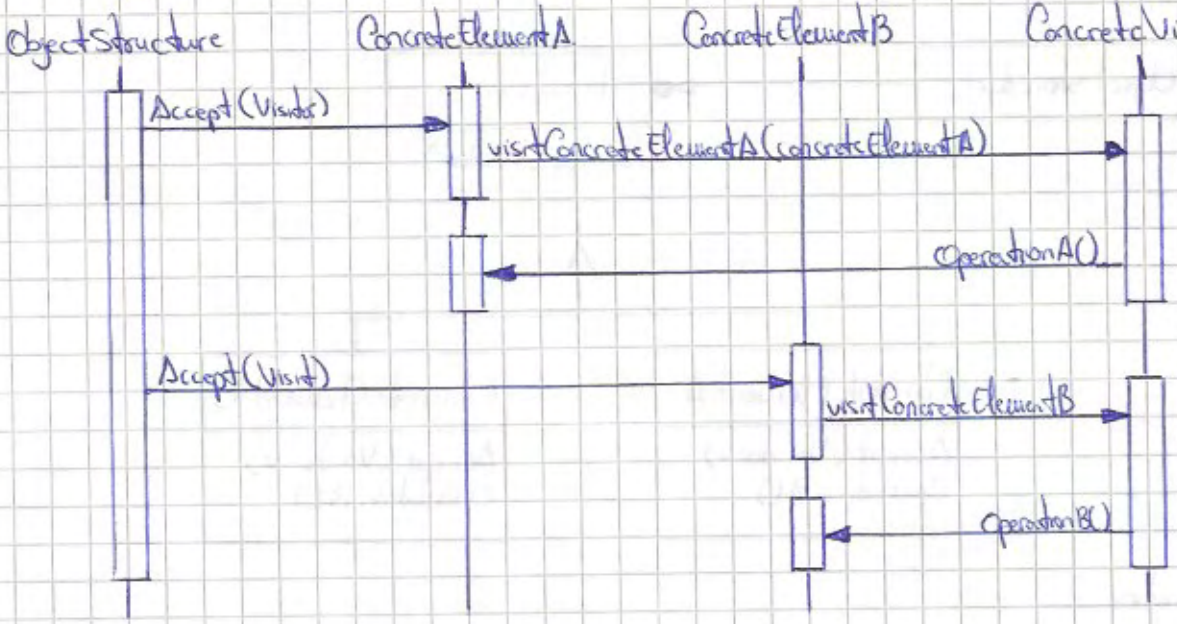


Participantes.

- Visitor.
  - Declara una operación visit para cada clase de operación ConcreteElement de la estructura de objetos.
  - El nombre y signatura de la operación identifican a la clase que envía la petición visit al visitante.
    - Eso permite al visitante determinar la clase concreta de elemento que está siendo visitada.
    - A continuación, el visitante puede acceder al elemento directamente a través de su interfaz particular.
- ConcreteVisitor.
  - Implementa cada operación declarada por Visitor.
  - Cada operación reimplementa un fragmento del algoritmo y guarda su estado local.
    - Muchas veces este estado acumula resultados durante el recorrido de la estructura.

- Element.
  - Define una operación accept que recibe una visitante como argumento.
- ConcreteElement
  - Implementa una operación accept que recibe un visitante como argumento.
- ObjectStructure.
  - Puede enumerar sus elementos.
  - Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.
  - Puede ser un compuesto (patrón Composite) o una colección, como una lista o un conjunto.

Colaboraciones
- Un cliente que usa el patrón Visitor debe crear un objeto ConcreteVisitor, y a continuación recorrer la estructura, visitando cada objeto con el visitante.
- Cada vez que se visita a un elemento, este llama a la operación del visitor que se corresponde con su clase.
  - El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.



Consecuencias.
- El visitante facilita añadir nuevas operaciones
  - Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante.
  - Si, por el contrario, extendiéramos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación.
- Un visitante agrupa operaciones relacionadas y separa las que no lo están.
  - El comportamiento similar no está desperdigado por las clases que definen la estructura de objetos; está localizado en un visitante.
  - Las partes de comportamiento no relacionadas se dividen en sus propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes.

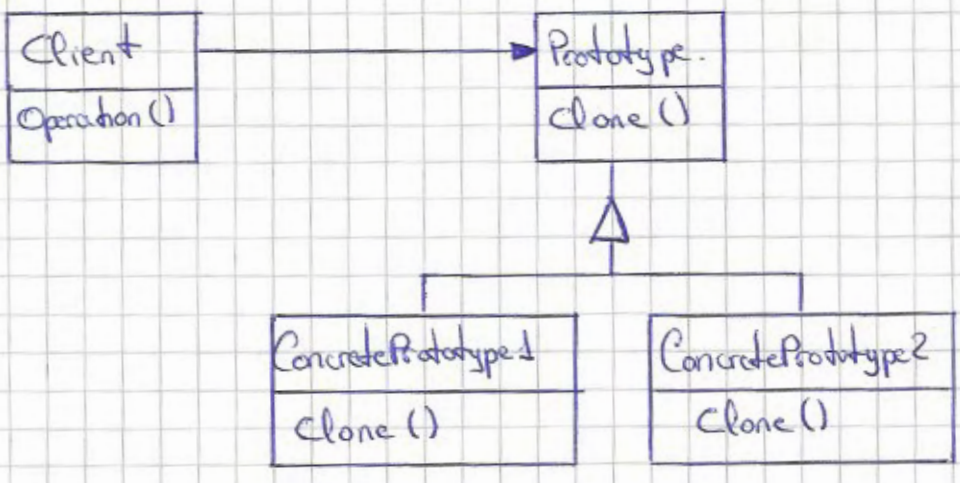
- Es difícil añadir nuevas clases de elementos concretos.
  - El patrón visitor hace que sea complicado añadir nuevas subclases de elementos.
    - Cada ConcreteElement nuevo da lugar a una nueva operación abstracta del Visitor y a su correspondiente implementación en cada clase ConcreteVisitor
    - A veces se puede proporcionar en Visitor una implementación predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esta representa una excepción más que una regla.
  - Por tanto la cuestión fundamental a considerar a la hora de aplicar el patrón Visitor es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura.

PROTOTYPE

Propósito.
Especifica los tipos de objetos a crear usando una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.

Aplicabilidad.
- Cuando un sistema no pueda (o no deba) conocer cómo se crean, componen y representan los productos.
- Las clases a instanciar son definidas en tiempo de ejecución.
- Para evitar construir una jerarquía paralela de factorías de productos.
- Cuando las instancias de una clase puedan tener sólo unos pocos posibles estados, y pueda resultar más conveniente crear los objetos correspondientes como prototipos y clonarlos, en vez de instanciar manualmente la clase, cada vez con el estado necesario.

Estructura.



Participantes.
- Prototype
  - Declara la interfaz (normalmente, una única operación) para clonarse.
- ConcretePrototype
  - Implementa la operación de clonación.
- Client
  - Crea un objeto diciéndole al prototipo que se clone.

Consecuencias.
- Como el patrón Abstract Factory oculta las clases concretas de producto al cliente.
- Añadir y eliminar productos dinámicamente (en tiempo de ejecución).
  - Simplemente registrando de algún modo el producto en el cliente.
- Especificar nuevos objetos modificando valores de sus propiedades
  - Mediante composición de objetos.
  - Es como si los usuarios creasen nuevas "clases" sin programar.
- Especificar nuevos objetos variando su estructura.
  - A partir de partes y subpartes.
  - Podemos guardar esas estructuras complejas para crearlas una y otra vez.
- Reduce las subclases
  - A diferencia del Factory Method, no hace falta una jerarquía paralela de clases de creación
- Inconvenientes.
  - La implementación de la operación de clonación puede no ser fácil.
    - Si las clases ya existieran
    - Si incluyen otros objetos que no se pueden clonar.
    - En presencia de referencias circulares.