Repaso de Patrones

> *Un patrón es la solución a un problema en un contexto. Cada patron describe un problema*
>
> *que ocurre repetidas veces y describe la solución a ese problema*

En general, un patron tiene 4 elementos escenciales :

- **Nombre del patron**
- **Problema :** Describe cuando aplicar el patron. Describe el problema y su contexto.
- **Solución :** Describe los elementos que constituyen el diseño, las relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o implementación concreta, porque un patron es como una plantilla que puede ser usada en multiples situaciones
- **Consecuencias :** Son los resultados e intercambios de aplicar el patron. Las consecuencias en un patron incluyen su impacto en la flexibilidad de un sistema, extensibilidad y portabilidad

Características

- **Nombre del patron y clasificación**
- **Propósito/intención** : una frase breve que responda las siguientes cuestiones
  - ¿Qué hace este patron de diseño?
  - ¿En qué se basa?
  - ¿Cuál es el problema concreto de diseño que resuelve?
- **Motivación** : un escenario/ejemplo que ilustra un problema de diseño y cómo las estructuras de clases y objetos del patron resuelven el problema.
- **Aplicabilidad** : ¿Cuando usamos el patron? ¿En qué situaciones se puede aplicar el patrón de diseño?¿Qué ejemplos hay de malos diseños que el patron puede resolver?
- **Estructura** : una representación grafica de las clases del patron
- **Participantes** : las clases y objetos participantes en el patrón de diseño, junto con sus responsabilidades
- **Colaboraciones** : Cómo colaboran los participantes para llevar a cabo sus responsabilidades
- **Implementación :** Qué trampas, pistas o tecnicas deberia tener presentes a la hora de implementar el patron? Hay problemas especificos segun el lenguaje?

Patrones Estructurales

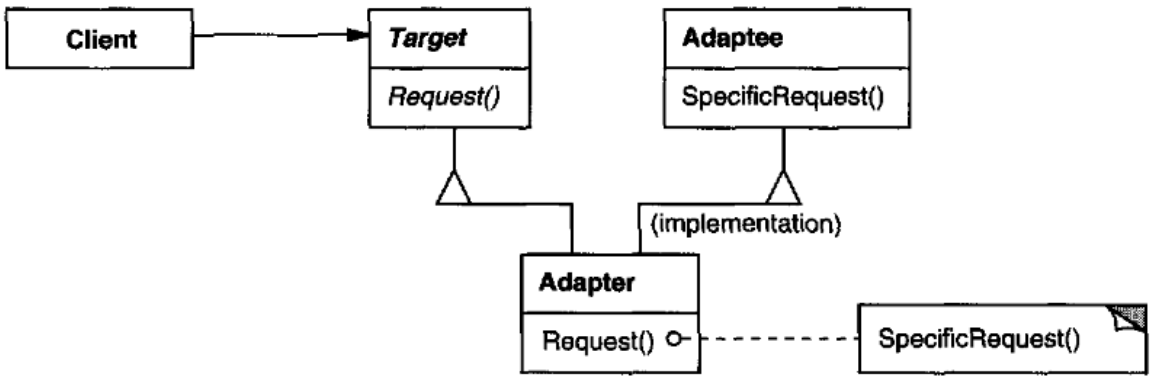
- Se preocupan de cómo se componen las clases y los objetos para formar estructuras grandes

Adapter

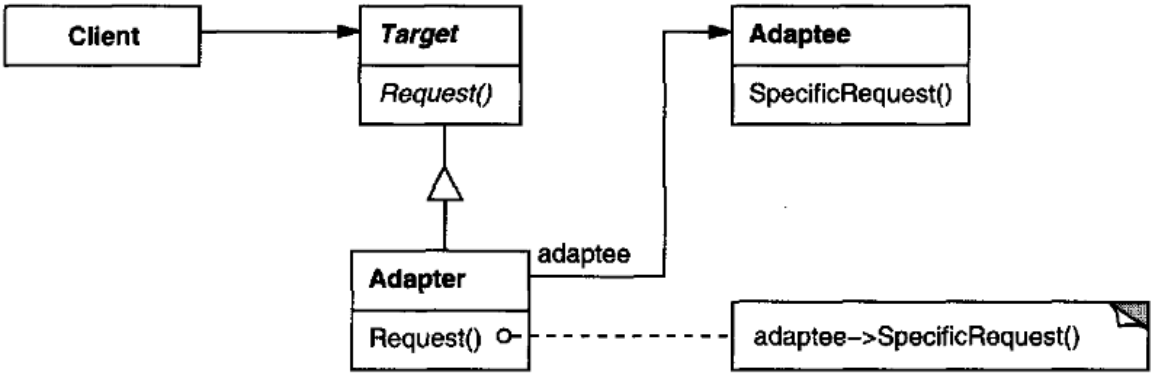
- Patron estructural

- **Propósito/Intención**
  - Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles
  - Convierte la interfaz de una clase en otra interfaz esperada por el cliente
- **Aplicabilidad (Se usa cuando...)**
  - Se quiere usar una clase y su interfaz no machea con la interfaz que yo necesito
  - Se quiere crear una clase reusable que copeere con clases no relacionadas o imprevistas (clases que no necesariamente tienen interfaces compatibles) → Adapter
- **Relación con otros patrones**
- **Estructura**
  - Opción 1 : La clase adapter usa multiples herencias para adaptar una interfaz a otra
  - Opción 2 : El objecto adapter depende de la composición de objetos

Opción 1



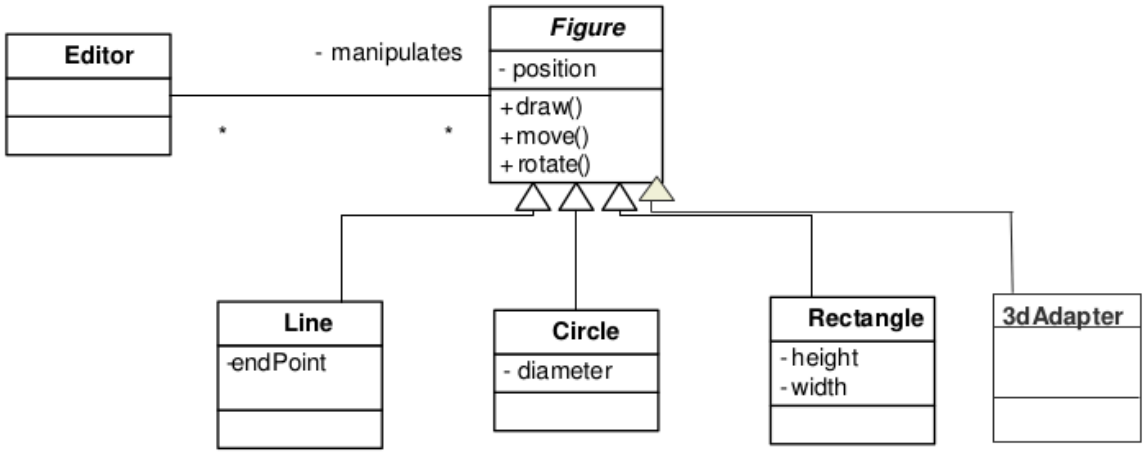
Opción 2



- **Target** : define la interfaz del dominio que usa el Cliente
- **Client** : colabora con objetos que se ajustan (implementan) a la interfaz Target
- **Adaptee** : define una interfaz existente que necesita ser adaptada. Client no puede utilizar directamente esta clase porque tiene una interfaz incompatible.
- **Adapter** : adapta la interfaz de Adaptee a la interfaz Target para que Client la pueda usar

Sobre la implementación :

- Como Adapter es subclase de Target, Adapter podrá reimplementar los métodos de su clase padre y a su vez la clase Adapter será tambien “una clase Target” (Adapter extiende a Target)
- Lo importante de esto es que Client podrá usar a los métodos de Target instanciando a Adapter, como se ve a continuación (Editor podrá instanciar a 3dAdapter como si fuera una figura cualquiera)



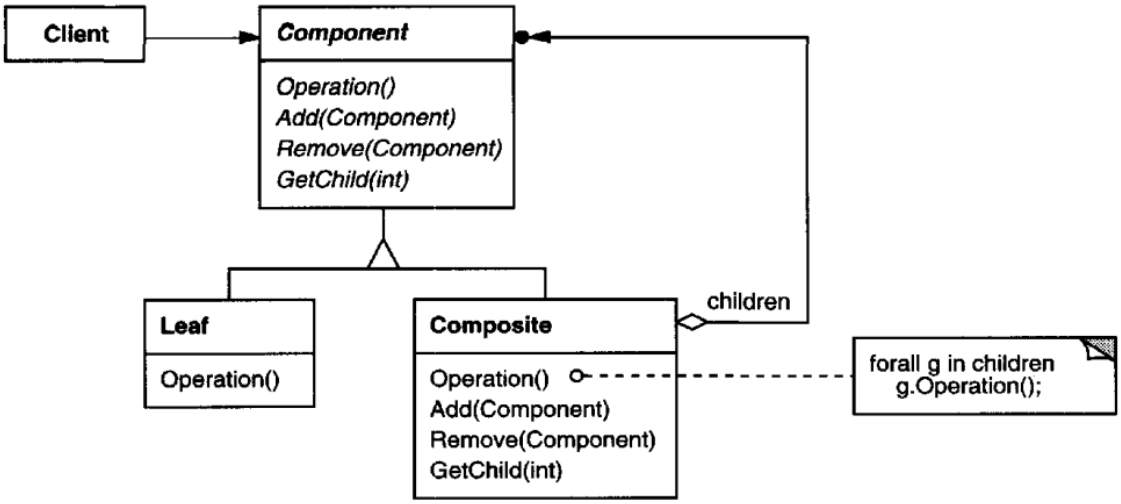
-

Composite

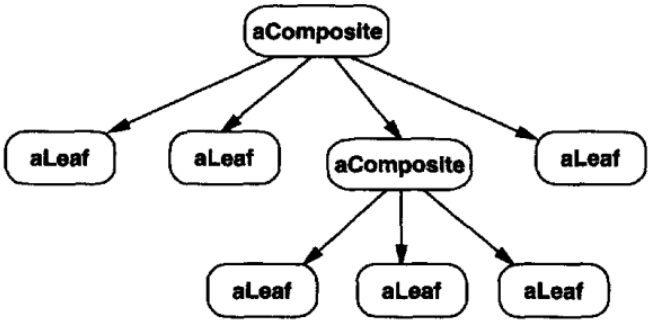
- Permite que un cliente trate a objetos atómicos y compuestos uniformemente

- **Propósito**
  - Componer objetos en estructuras de árbol para representar jerarquías parte-todo
    - La clave acá esta en la composición
    - Se componen objetos para formar estructuras de arbol que representen jerarquias, y así tratar a los objetos atomicos y composiciones de igual manera

    - Composite permite que los clientes traten a los objetos atómicos y a sus composiciones uniformemente
- **Aplicabilidad (Se usa cuando...)**
    - Se quiere que los objetos “clientes” puedan ignorar las diferencias entre composiciones y objetos individuales. Los clientes trataran a todos los objetos en la estructura compuesta de manera uniforme
    - Se quiere representar jerarquías parte-todo de objetos
- **Relación con otros patrones**
    - Se usa Builder cuando se crean arboles Composite complejos porque podes programar los pasos de construcción para que trabajen recursivamente
    - Composite y Decorator tienen estructuras similares, ambos usan la composición recursiva
- **Consecuencias**
    -
- **Participantes :**
    - **Component :** Declara la interfaz para los objetos de la composición. Implementa comportamientos default para la interfaz común a todas las clases. Declara la interfaz para definir y acceder a los hijos
    - **Leaf :** Representa arboles “hojas” en la composición. Las hojas no tienen hijos. Define el comportamiento de objetos primitivos en la composición
    - **Composite :** Define el comportamiento para componentes con hijos. Contiene la referencia a los hijos. Implementa operaciones para manejar hijos



A typical Composite object structure might look like this:



Sobre la implementación :

- Tanto las partes simples (leaf) como las composiciones (Composite) responden al mismo protocolo que lo podrá determinar una interfaz o una clase abstracta
- Observar que es medio “recursivo” porque composite (una composición) puede conocer tanto a una leaf como a otra composición. Por eso se habla

que el patron tiene una estructura de árbol

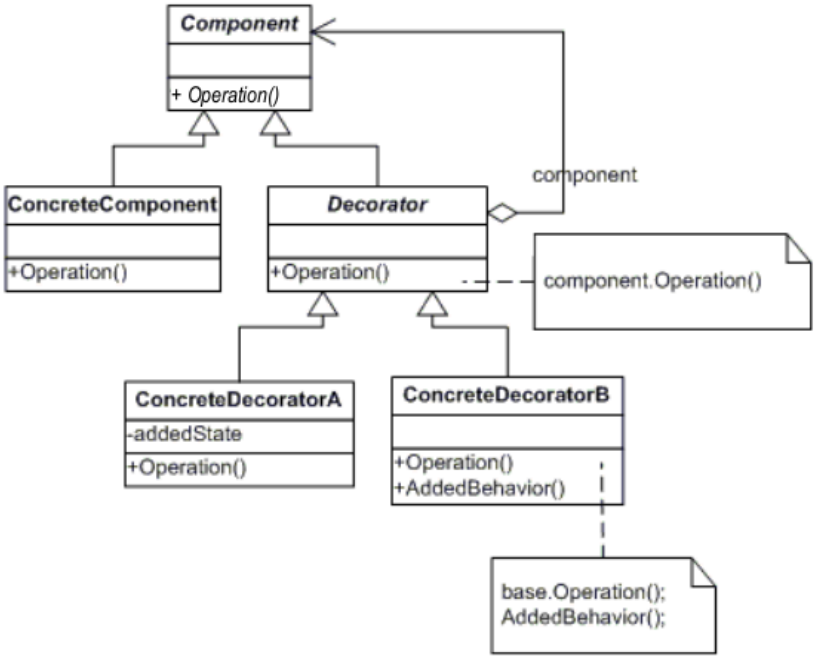
- El mensaje que recibe composite lo deben ejecutar todos los hijos
- Sería importante redefinir métodos en las hojas para no tener errores. Ya que por ejemplo, leaf no deberia devolver nada al recibir el mensaje getChild()
- En general, el rol Composite conoce a muchos Component (caso contrario al Patron Decorator)

Decorator

- **Proposito**
  - Agregar comportamiento a un objeto dinámicamente y en forma transparente
  - Asigna responsabilidades adicionales a un objeto dinamicamente, proporcionando una alternativa flexible a la herencia para extender funcionalidad
- **Aplicabilidad**
  - Para agregar responsabilidades a objetos individuales dinamicamente y de forma transparente (sin afectar a otros objetos)
  - Para responsabilidades que pueden ser retiradas
  - Cuando la extension mediante la herencia no es viable
- **Relación con otros patrones**
  - Decorator puede ser visto como un Composite degenerado, que cuenta unicamente con un componente. Sin embargo, Decorator añade responsabilidades adicionales
  - En relación al patron Strategy, Decorator permite cambiar “la piel” de un objeto, mientras que Strategy permite cambiar “las entrañas”
- **Problema que resuelve**

- Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”. El problema que tiene la herencia es que se decide estáticamente

- **Motivación**
  - A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase



Sobre la implementación

- Misma interface entre Component y Decorator
- No hay necesidad de la clase Decorator abstracta
- Un Component puede ser 2 cosas : un componenteConcreto o un Decorador
  - El Component en si es una interfaz

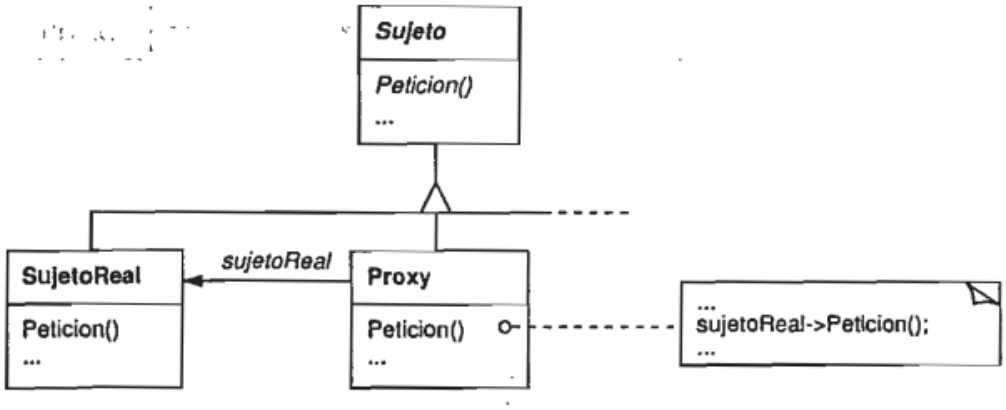
- Lo que realmente se instancia podrá ser un Decorador o un ConcreteComponent
- La idea es que el Decorador “envuelve” a otro decorador o finalmente a un componente concreto
- Las subclases de Decorador son libres de añadir operaciones para determinadas funcionalidades

Proxy

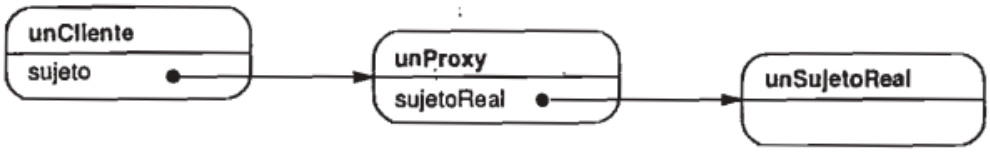
- **Proposito**
  - Proporcionar un intermediario de un objeto para controlar su acceso
  - Propociona un representante o sustituto de otro objeto para controlar el acceso a este
- **Aplicabilidad**
  - Virtual proxy : se usa cuando se tiene un objeto muy pesado que consume muchos recursos, entonces se retrasa su instanciacion unicamente hasta que se lo necesita
  - Protection proxy : se usa cuando se quiere que solo ciertos cliente accedan al servicio real
  - ...
- **Relación con otros patrones**
  - Con Adapter, es posible acceder a un objeto existente a traves de una interfaz diferente. Con Proxy, la interfaz es la misma. Con Decorator accedes al objeto a traves de una interfaz mejorada
  - Decorator y Proxy tienen estructuras similares, pero propositos muy distintos. Ambos se basan en el principio de composición, donde un objeto se supone que delega trabajo en otro. La diferencia esta en que

Proxy generalmente maneja el ciclo de vida de su objeto servicio, mientras que Decorator es siempre controlado por el cliente

- **Estructura**



Éste es un posible diagrama de objetos de una estructura de proxies en tiempo de ejecución:



Sobre la implementación

- Proxy es un objeto intermedio que respeta el protocolo/mantiene el protocolo del objeto que esta reemplazando

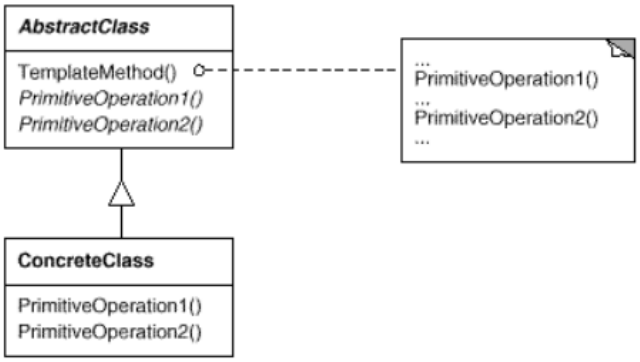
Patrones de Comportamiento

Template method

- **Propósito**

    - Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos.
    - Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura
- **Aplicabilidad (Se usa cuando...)**
    - Para implementar partes invariantes de un algoritmo una sola vez y dejarlo asi para que subclases implementen el comportamiento que puede variar
    - Cuando comportamiento comun entre subclases debe ser localizado en una clase comun para evitar la duplicación de codigo (refactorizar para generalizar)
    - Para controlar la extensión de subclases. Se puede definir un template method que llame a operaciones “hook” / “ganchos” en puntos especificos, permitiendo así la extensión sólo en esos puntos
- **Consecuencias**
    - El template method lleva a una estructura de control invertida, en la cual, la clase padre llama a las operaciones de la subclase y no de la otra manera
    - Los template method llaman a los siguientes tipos de operaciones :
        - Operaciones concretas (en la clase concreta o en la clase cliente)
        - Operaciones abstractas concretas (operaciones que deben implementar las subclases)
        - Operaciones primitivas (operaciones abstractas) - ???
        - Factory methods - ???
        - **Operaciones hook,** las cuales proveen un comportamiento default que las subclases extienden si es necesario. Un hook generalmente no hace nada por default
    - Es importante para un template method especificar qué operaciones son hooks (**pueden** ser sobreescritas) y cuales son operaciones abstractas (**deben** ser sobreescritas)
- **Ejemplos**

- Rep



Sobre la implementación :

- La clase abstracta que define el esqueleto podria tambien asignar un comportamiento “default” a las operaciones primitivas
- ¿Qué pasará cuando a ConcreteClass le llegue un mensaje TemplateMethod?
  - Como ConcreteClass no posee dicho método, lo buscara en la superclase (**Method lookup**)
  - Sin embargo, cuando dentro de TemplateMethod se llame con this a una de las operaciones primitivas, lo que se ejecutará es la operación primitiva de ConcreteClass en caso de estar implementada.

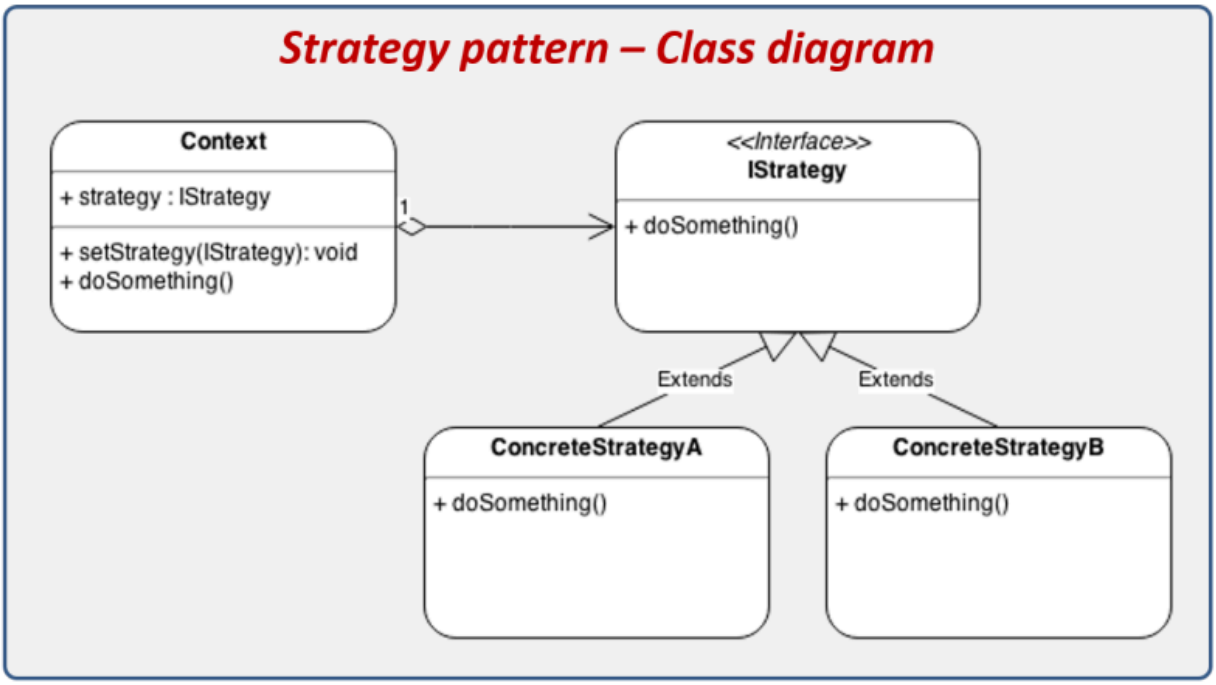
Strategy

- Permite cambiar el algoritmo que un objeto utiliza en forma dinámica

- **Propósito**

- Define una familia de algoritmos, encapsula cada uno y los vuelve intercambiables.
- El Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan.
- Los clientes pueden cambiar el algoritmo que están utilizando sin necesidad de modificar su propio código

- **Aplicabilidad (Se usa cuando...)**
  - Muchas clases relacionadas difieren solo en el comportamiento. Strategy provee una manera de configurar una clase con uno de muchos comportamientos
  - Se necesitan distintas variantes de un algoritmo / Existen muchos algoritmos para llevar a cabo una tarea
  - Un algoritmo usa información que el cliente no deberia conocer / Cada algoritmo utiliza información propia
  - Una clase define muchos comportamientos y estos aparecen como multiples sentencias condicionales en sus operaciones.



- **Relación con otros patrones**
  - Strategy se relaciona con los patrones Decorator, Template method y State
  - Decorator permite cambiar la “piel” de un objeto mientras que Strategy permite cambiar “las tripas”. Decorator actúa como una "piel" o una envoltura que proporciona nuevos comportamientos de manera externa. Por ejemplo : Si tienes una clase Coffee, un MilkDecorator podría envolverla para agregar leche sin modificar la clase Coffee directamente. Por otro lado, Strategy modifica el comportamiento interno de un objeto, permitiendo reemplazar algoritmos sin cambiar la estructura del objeto. Por ejemplo : Si tienes una clase PaymentProcessor, podrías usar diferentes estrategias (CreditCardPayment, PayPalPayment) para definir cómo se realiza un pago sin modificar PaymentProcessor
  - Template method se basa en la herencia, mientras que Strategy se basa en la composición. Template method permite alterar partes de un algoritmo extendiendo esas partes en subclases. Mientras que Strategy permite alterar el comportamiento de un objeto dandole estrategias que se ajusten a ese comportamiento. Template method funciona a nivel de clase,

es estatico. Strategy trabaja a nivel de objetos, permitiendo cambiar comportamiento en tiempo de ejecución.

- Con respecto al patron State, ambos se basan en la composición. Ambos cambian el comportamiento del contexto, delegando trabajo en otros objetos. Strategy hace a estos objetos independientes e inconscientes unos de otros. State no restringe la dependencia entre estados concretosc, permitiendoles cambiar el estado del contexto a voluntad

Sobre la implementación :

- La clase Strategy puede ser interfaz o clase abstracta
- En general, el cliente CREA una estrategia en concreto, la instancia, y se la manda al contexto. Pero cliente no conoce en si a la estrategia, no hay una relación (Entiendo que no lo tiene como variable de instancia, una cosa asi será)

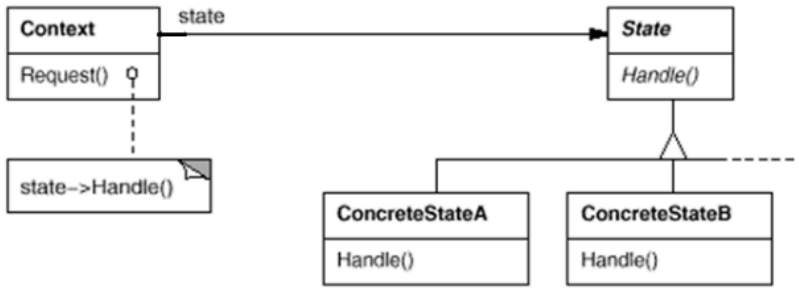
State

- Lo usamos cuando el comportamiento de un objeto depende del estado en el que se encuentre
- El patron State si o si genera acomplamiento y puede terminar exponiendo el estado interno de una clase

- **Propósito**
  - Modificar el comportamiento de un objeto cuando su estado interno se modifica
  - Externamente parecería que la clase del objeto ha cambiado
- **Aplicabilidad (Se usa cuando...)**

    - El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de su estado
    - Los métodos tienen sentencias condicionales complejas que dependen del estado
    - Este estado se representa usualmente por constantes enumerativas y en muchas operaciones aparece el mismo condicional (Por ejemplo, en varios métodos de una clase tengo algo como `if ( color == rojo ) { ... }`
    - El patron State reemplaza el condicional por clases (es un uso inteligente del polimorfismo). Ya que internamente estoy reemplazando en tiempo de ejecución un objeto por otro (cada objeto representa un estado)
- **Participantes**
    - **Context :** Define la interfaz que conocen los clientes (Podria ser la interfaz alarma por ejemplo). Mantiene una instancia de alguna clase de ConcreteState que define el estado corriente. Quien tiene ESTADOS es el contexto
    - **State :** Define la interfaz para encapsular el comportamiento de los estados de Context
    - **ConcreteState :** Cada subclase implementa el comportamiento respecto al estado especifico
- **Consecuencias**
    - En general hay bastante acoplamiento entre las subclases de State porque la transición de estados se hace entre ellas, por lo que deben conocerse entre sí

- Estructura



Sobre la implementación

- ¿Quien define la transición entre estados? El patron state no especifica qué participante define los criterios para las transiciones entre estados. En general, es más conveniente que sean las propias subclases de State quienes especifiquen su estado sucesor y cuándo llevar a cabo la transición. Esto requiere añadir una interfaz al Contexto que permita a los objetos estado asignar explicitamente el estado actual del Contexto (meter un seter al Context, lo cual rompe el encapsulamiento)
- Dos opciones :
  - Crear los objetos Estado sólo cuando se necesitan y destruirlos despues
    - Esta opcion es preferible cuando no se conocen los estados en tiempo de ejecucion y los contextos cambian de estado con poca frecuencia
  - Crearlos al principio y no destruirlos nunca
    - Este enfoque es mejor cuando los cambios tienen lugar repentinamente, en cuyo caso queremos evitar destruir los estados, ya que pueden volver a necesitarse de nuevoe n breve.
    - Este enfoque puede no ser apropiado ya que el Contexto debe guardar referencias a todos los estados en los que pudiera entrar

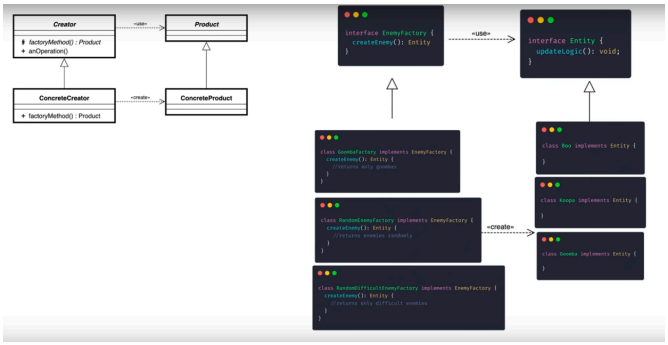
- En la practica, State es una clase Abstracta/Interfaz y debe implementar todos los metodos del contexto
- Si el contexto tiene :
  - Abrir
  - Cerrar
  - Entonces State deberá tener los mismos métodos
  - Las subclases concrectas tambien deberán implementar estos métodos

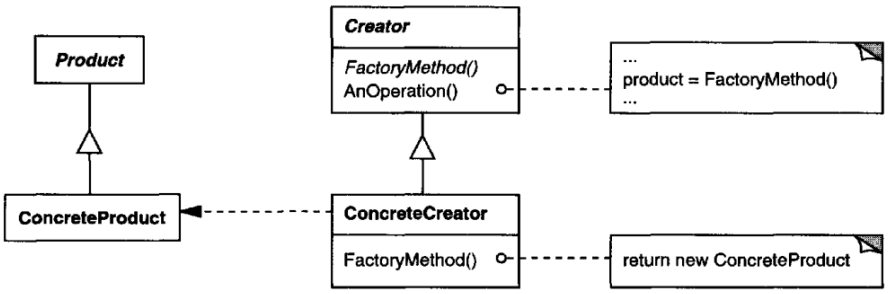
Patrones de Creación

Factory method

- **Proposito**
  - Define una “interfaz” para la creación de objetos, pero permite que subclases decidan qué clase se debe instanciar (Ver ejemplo de la derecha)
- **Aplicabilidad**
  - Una clase no puede anticipar que clase de objeto debe crear
  - Una clase quiere que sus subclases especifiquen el objeto a crear
  - Usar cuando se quiera proveer a los usuarios de un framework o libreria una forma de extender sus componentes internos
- **Relacion con otros patrones**
  - Los Factory Methods son una especialización el template method, donde lo que se delega es la instanciación de un objeto

- Estructura





Sobre la implementación

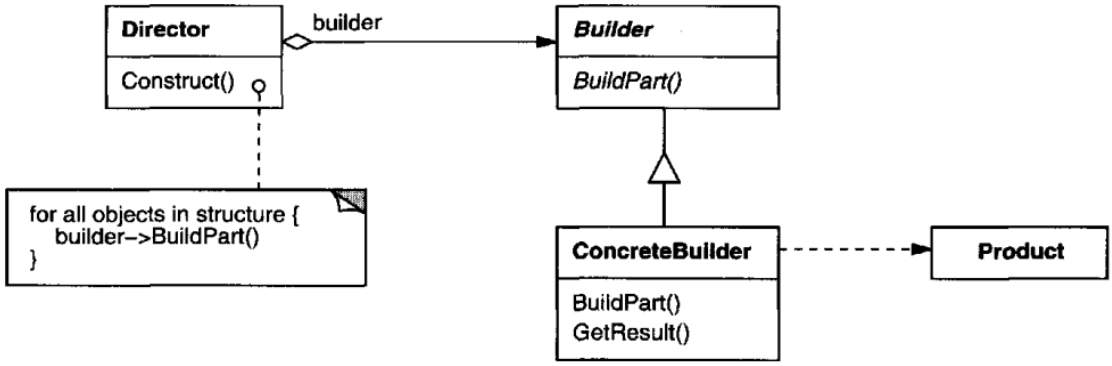
- Dos variantes principales. Las dos principales variantes del patrón Factory Method son :
  - Cuando la clase Creador es una clase abstracta y no proprociona una implementación para el método de fabricación que declara. Este caso requiere que las subclases definan una implementación porque no hay ningun comportamiento predeterminado razonable
  - Cuando el Creador es una clase concreta y proporciona una implementación predeterminada del método de fabricación
- Métodos de fabricación parametrizados. Otra variante del patrón permite que los métodos de fabricación creen varios tipos de productos. El método de

fabricación recibe un parametro que identifica el tipo de objeto a crear. Todos los objetos creados por el método compartiran Producto

Builder

- **Propósito**
  - Separa la construcción de un objeto complejo de su representación de manera que el proceso de construcción pueda crear diferentes representaciones
  - Permite construir un objeto complejo paso a paso. Permite producir diferentes tipos y representaciones de un objeto usando el mismo codigo
- **Aplicabilidad (Se usa cuando...)**
  - El algoritmo de creación de un objeto complejo deberia ser independiente de las partes que hacen al objeto y cómo se ensamblan
  - El proceso de construcción debe permitir diferentes representaciones del objeto que es construido
- **Estructura**
  - **Builder** : Especifica una interfaz abstracta para crear partes de un objeto Producto (Declara los pasos de construcción del producto)
  - **ConcreteBuilder** :
    - Construye y ensambla partes del Producto implementando la interfaz de Builder
    - Provee una interfaz para recuperar Producto
    - Provee diferentes implementaciones de los pasos de construcción. Estos pueden producir productos que no siguen la interfaz comun
  - **Director** :
    - Construye un objeto usando la interfaz Builder
  - **Product** :

- Representa el objeto completo bajo construcción



- Por qué se dice que el patron builder permite construir diferentes representaciones de un objeto?
  - El patrón **Builder** permite construir **diferentes representaciones** de un mismo objeto porque separa **el proceso de construcción** de la representación final del objeto
  - Esto significa que con el mismo **Builder** podemos crear variaciones del **Product**, dependiendo de qué datos o configuraciones le demos.

https://refactoring.guru/es/design-patterns/builder

Ejemplo : Construir una casa

- Puedes crear un enorme constructor dentro de la clase base `Casa` con todos los parámetros posibles para controlar el objeto casa.
- En la mayoría de los casos, gran parte de los parámetros no se utilizará, lo que provocará que las llamadas al constructor sean bastante feas. Por ejemplo, solo una pequeña parte de las casas tiene piscina, por lo que los parámetros relacionados con piscinas serán inútiles

- El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados *constructores*
- Podemos crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente. Entonces podemos utilizar estos constructores en el proceso de construcción (por ejemplo, una serie ordenada de llamadas a los pasos de construcción) para producir distintos tipos de objetos
- La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos
    - No es estrictamente necesario tener una clase directora en el programa, ya que se pueden invocar los pasos de construcción en un orden específico directamente desde el código cliente. No obstante, la clase directora puede ser un buen lugar donde colocar distintas rutinas de construcción para poder reutilizarlas a lo largo del programa.

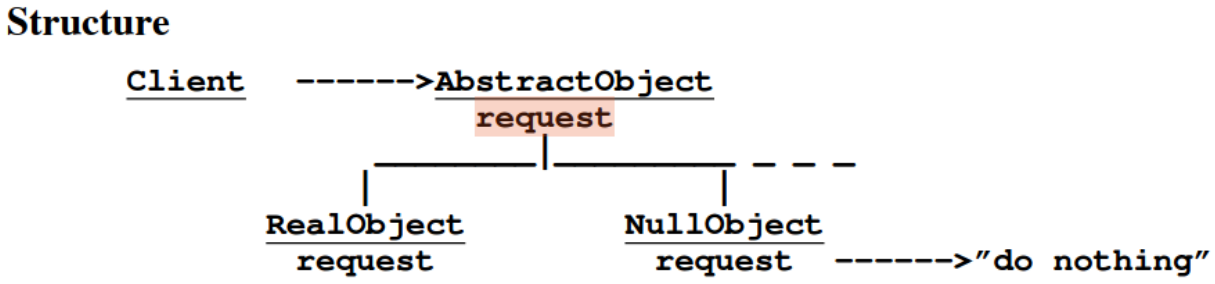
Los distintos constructores ejecutan la misma tarea de formas distintas.

Patrones de Dominio

Null object

- **Proposito**
  - Proporciona un sustituto para otro objeto (similar a proxy) que comparte la misma interfaz pero no hace nada.
  - El NullObject encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores
- **Aplicabilidad**
  - Se usa cuando un objeto requiere un colaborador. Esta colaboración ya existia implicitamente, no es que el patron Null Object “la introduce”
- **Consecuencias**
  - Elimina todos los condicionales que verifican si la referencia a un objeto es NULL
  - Hace explícito elementos del dominio que hacen “nada”
- **Participantes**
  - Client
  - AbstractObject :
    - Declara la interfaz que usará Client
    - Implementa comportamiento por defecto
  - RealObject :
    - Define una subclase concreta de AbstractObject cuya instancia provee comportamiento útil para el cliente
  - NullObject
    - Provee una interfaz identica a AbstractObject de manera que un null object puede ser sustituido por un real object

        - Implementa la interfaz para no hacer nada. El hacer nada depende del comportamiento que Client este esperando
    - **Relación con otros patrones**
        - NullObject es tipicamente usado como una clase en la jerarquia de un Strategy. Representa la estrategia para no hacer nada
        - Al igual que con Strategy, sucede lo mismo con State



Type object

- **Proposito**
- **Aplicabilidad**
    - Se requiere agrupar instancias de una clase según atributos y comportamiento comunes
    - Hay demasiadas subclases o un número desconocido de ellas.

Comparación entre Patrones

Proposito		
Cómo funciona		
Uso principal		
Cambio dinámico?		
Ejemplo típico		

Composite vs Decorator

	Composite	**Decorator**
Proposito	Permitir componer objetos para que se estructuren en forma de árboles jerárquicos. Para asi tratar a los elementos atomicos y a las composiciones de manera uniforme	Permitir añadir nuevas funcionalidades a un objeto (agregando capas) sin modificar su código base, envolviéndolo dinámicamente
Ejemplo típico	Sistema de archivos y carpetas. Sistema de Paneles con Botones, Etiquetas y otros paneles	Sistema de armado de cafés con toppings, Sistema de texto que permite agregar decoraciones al texto, Sistema de personajes con armaduras, armas o poderes

State vs Strategy

	State	Strategy
Proposito	Permitir que un objeto cambie su comportamiento dinámicamente según su estado interno	Permitir que un objeto utilice diferentes algoritmos intercambiables sin cambiar su estructura.
Cómo funciona	Define una serie de estados como clases separadas	Define una serie de estrategias (algoritmos) como clases separadas

Uso principal	Modelar objetos con múltiples estados internos y transiciones entre ellos.	Modelar objetos con diferentes formas de ejecutar una tarea
Cambio dinámico?	Sí: Cambia de estado automáticamente según el flujo	Sí: Puede cambiar de estrategia en tiempo de ejecución, pero no de forma automática, ya que necesita de un cliente que setee el algoritmo a usar
Ejemplo típico	Maquina expendedora : Estados: Sin monedas, Con monedas, Producto seleccionado → maneja transiciones como insertar moneda o elegir producto Sistema de autenticación : Estados: Usuario no logueado, Usuario logueado, Admin → cambian los permisos y opciones disponibles.	Sistemas de navegación: Estrategias: Ruta más rápida, Ruta más corta, Ruta turística → el usuario elige cómo calcular el trayecto. Metodos de pago: Estrategias: Tarjeta de crédito, PayPal, Criptomonedas → se aplica la lógica específica de cada medio Compresión de archivos: Estrategias: `ZIP`, `RAR`, `7z` → el sistema elige el algoritmo según necesidades.

Builder vs Factory Method

	Factory Method	**Builder**
Proposito	Define una “interfaz” para la creación de objetos, pero permite que subclases decidan qué clase se debe instanciar	Construir objetos complejos paso a paso
Uso principal	Cuando se necesita delegar la creación de instancias en subclases	Cuando un objeto tiene muchos atributos opcionales o configuraciones

Extensibilidad	Se basa en herencia y permite agregar nuevas implementaciones	Se basa en composición y permite construir objetos de manera flexible.

Adapter vs decorator

Ambos son patrones estructurales, y ambos de alguna manera “envuelven” a otro objeto

	Adapter	**Decorator**
Proposito	Hacer que una interfaz incompatible sea compatible con otra sin modificar el código original.	Agregar funcionalidades dinámicamente a un objeto (como aplicando capas) sin modificar su estructura original.
Cómo funciona	Traduce una interfaz a otra esperada por el cliente.	Envuelve un objeto y extiende su comportamiento.
Modifica la interfaz?	Sí: Transforma la interfaz del objeto adaptado para que sea compatible con la interfaz del cliente	No: Mantiene la interfaz del componente y solo le añade comportamiento
Ejemplo típico	Adaptar una API antigua a una nueva	...

Discusiones sobre patrones estructurales

Composite vs Decorator vs Proxy

- Composite y Decorator tienen diagramas de estructura similares, ya que ambos dependen de la **composición recursiva** para organizar un buen numero de objetos. Más allá de esto, no tienen más similitudes porque sus propositos son bien distintos
- En la implementación, tanto Decorator como Proxy mantienen una referencia a otro objeto al cual le envian mensajes. En el patron Proxy, el sujeto define la funcionalidad clave y el Proxy permite o deniega el acceso a este, mientras que en Decorator, el componente solo provee parte de la funcionalidad, y uno o más decoradores decoran el resto.

Dicusiones sobre patrones creacionales

(Builder y Factory method)

Hay dos formas comunes de parametrizar un sistema según las clases de objetos que crea :

- Una es creando subclases de la clase que crea los objetos : Factory method
- La otra manera, recae más en la composición de objetos

Dicusiones sobre patrones de comportamiento

- Encapsular una variación es un tema recurrente en los patrones de comportamiento

    - Un nuevo estado? → Lo encapsulo con State
    - Nueva estrategia? → Uso strategy
- Cuando un aspecto de un programa cambia frecuentemente, estos patrones definen un objeto que encapsula este aspecto