Unidad 5
Uso de Patrones de Diseño en el Paradigma Orientado a Objetos

Prof: Lic. Rosales Pablo (prosales@unpata.edu.ar)
Materia: Programación Orientada a Objetos
Año: 2024

Temario

- Patrones de diseño
- Creational patterns
  - Builder
  - Factory method
  - Singleton
- Structural patterns
  - Composite
  - Decorator
  - Facade
- Behavioral patterns
  - Observer
  - State
  - Strategy

Patrones de diseño

- Representan las mejores prácticas utilizadas por desarrolladores de SW orientado a objetos
- Son las soluciones a los problemas generales.
- Fueron obtenidas por prueba y error durante un tiempo considerable.

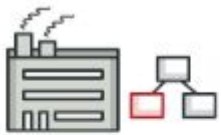
LIBROS:

Design Patterns - Elements of Reusable Object-Oriented Software, 1994, de Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides

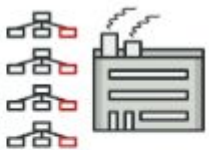
Dive Into Design Patterns, 2019, Alexander Shvets.

Creational patterns

Estos patrones de diseño proveen una forma de crear objetos escondiendo la lógica de creación (suplantando la utilización del new). Esto da a los programas más flexibilidad en decidir qué objetos necesitan ser creados para ciertas situaciones.

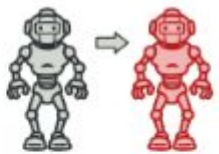


Factory Method



Abstract Factory

Builder



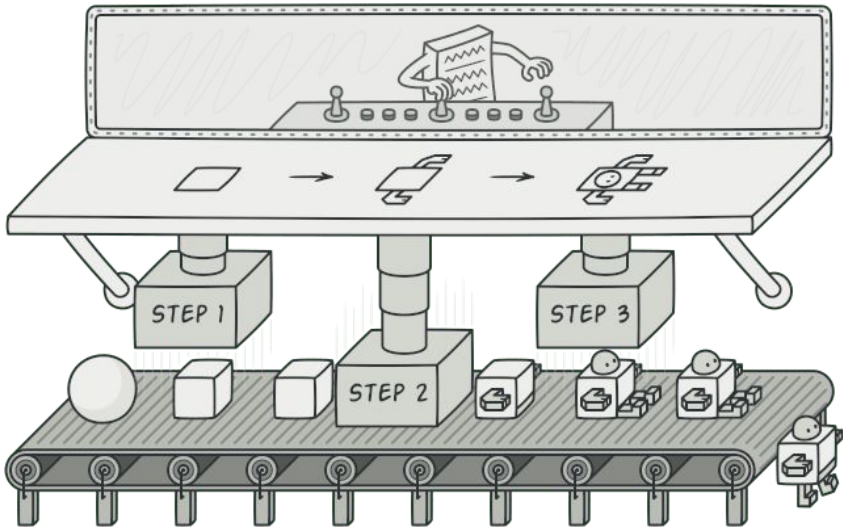
Prototype

Singleton

Builder

Parte de los patrones de creacional.

Nos permite construir objetos complejos paso a paso, en los cuales los objetos pueden tener diferentes representaciones usando el mismo código.



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

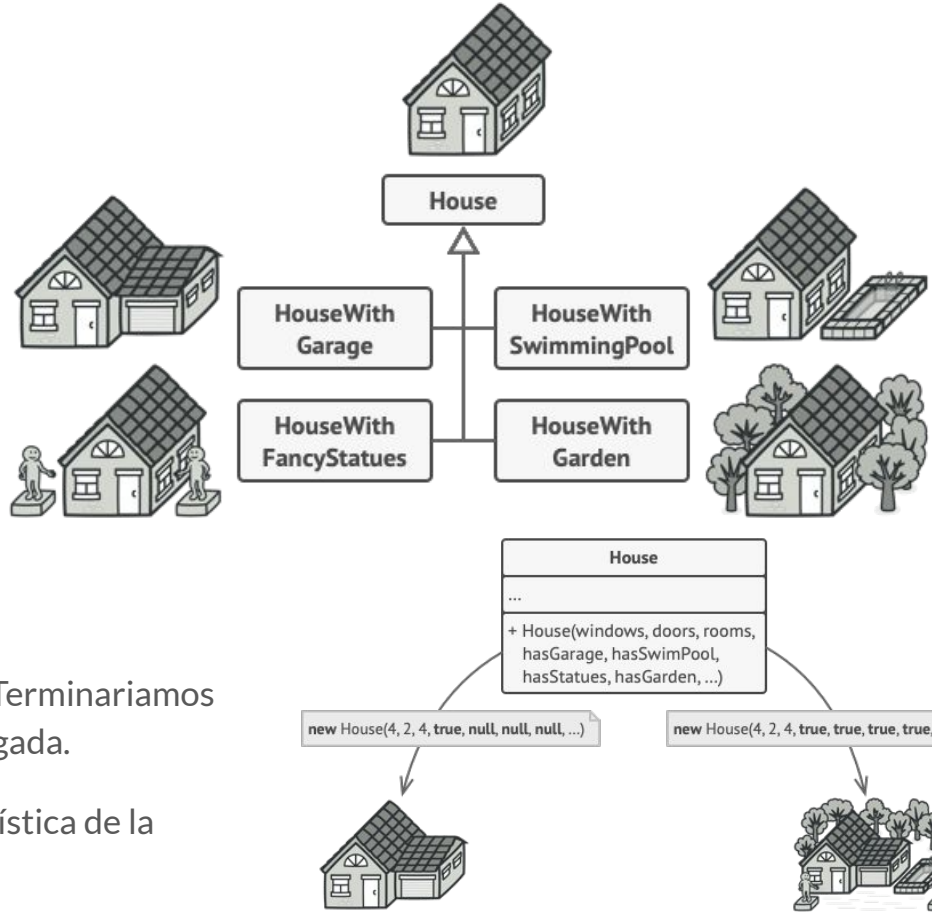
Problema

Cuando se tiene un objeto complejo que requiere de muchos pasos para inicializar sus atributos y objetos anidados, teniendo un constructor gigante con muchos parámetros.

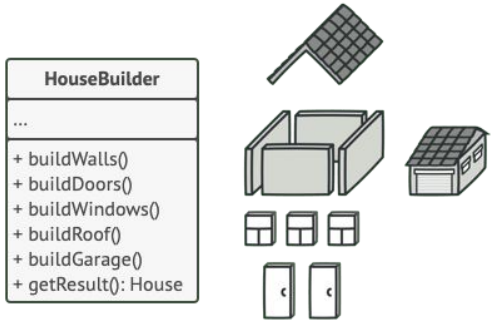
Para el ejemplo de la imagen:

¿Creamos una estructura de herencia de clases? Terminariamos con muchas subclases por cada mínima cosa agregada.

¿Una sola clase con parámetros por cada característica de la casa?



Solución



Extraer el código de la construcción del objeto fuera de su clase y moverlo a objetos Builder.

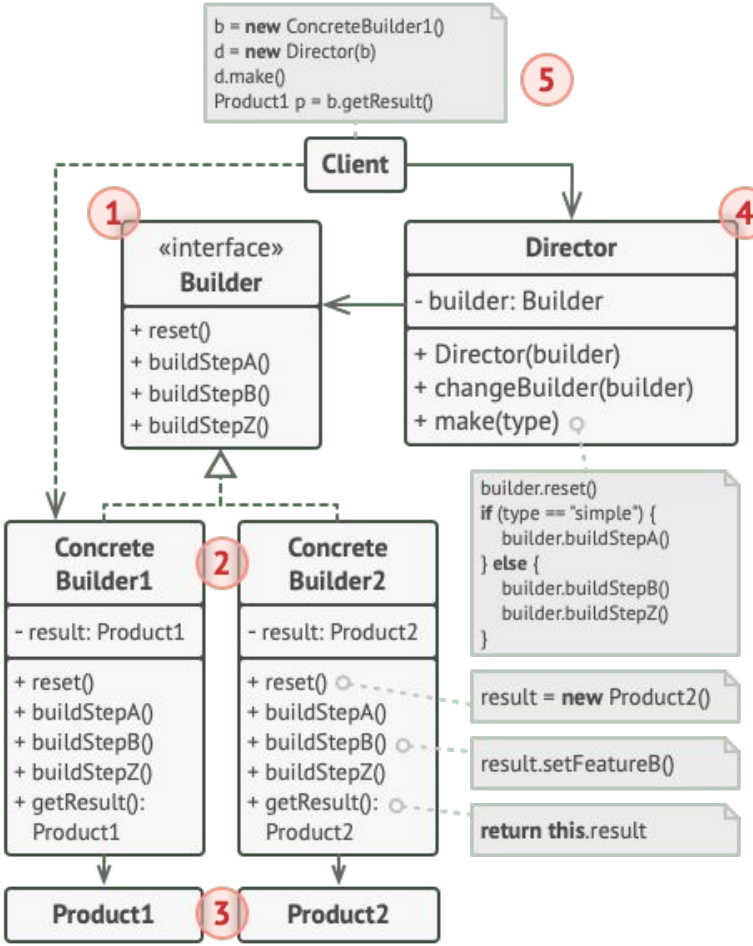
1- La interfaz **Builder** declara los pasos comunes a todos los builder.

2- Los **ConcreteBuilder** proveen las implementaciones.

3- **Product** el objeto a construir en cuestión.

4- **Director** define el orden de llamado de los pasos.

5- **Client** les da al director los Builder que va a usar.

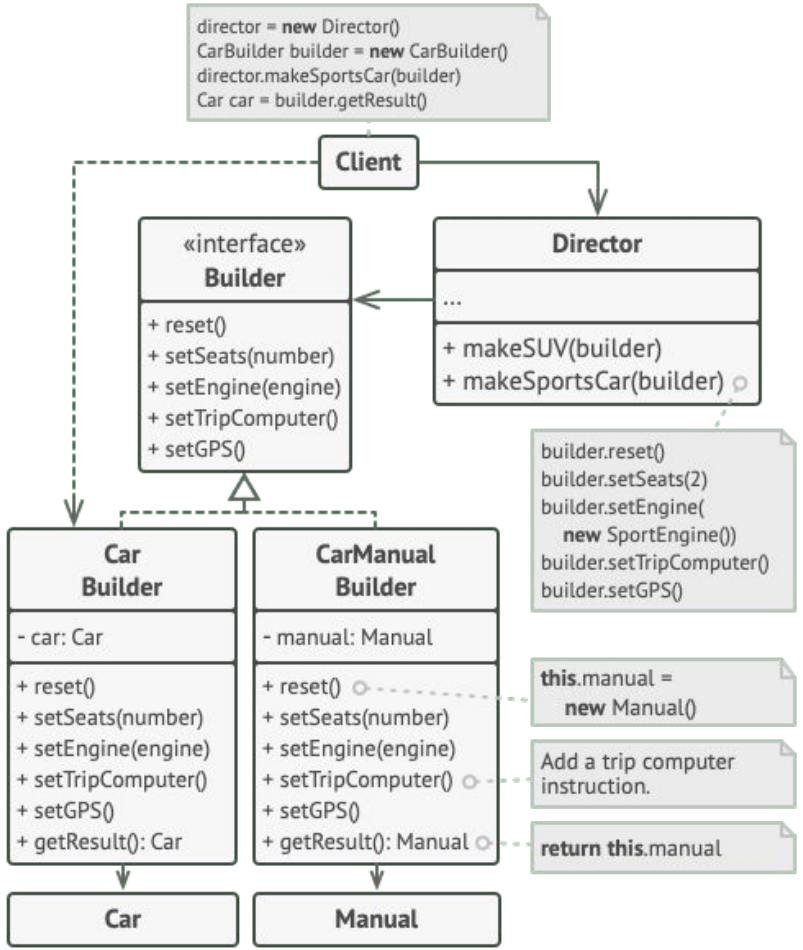


Ejemplo

El ejemplo apunta a crear diferentes tipos de autos como también algo tan diferente como un manual de usuario para un auto.

El Director controla el orden de la construcción, sabe los pasos a llamar para cada tipo de auto. Trabaja usando los builder que comparten una interfaz en común.

El resultado se obtiene desde el builder, porque el director no puede saber el objeto resultante. Solo el objeto Builder conoce que es lo que produce.



Usar cuando:

El algoritmo para crear el objeto complejo debe ser independiente de las partes que hacen al objeto y como debe ser “ensamblado”.

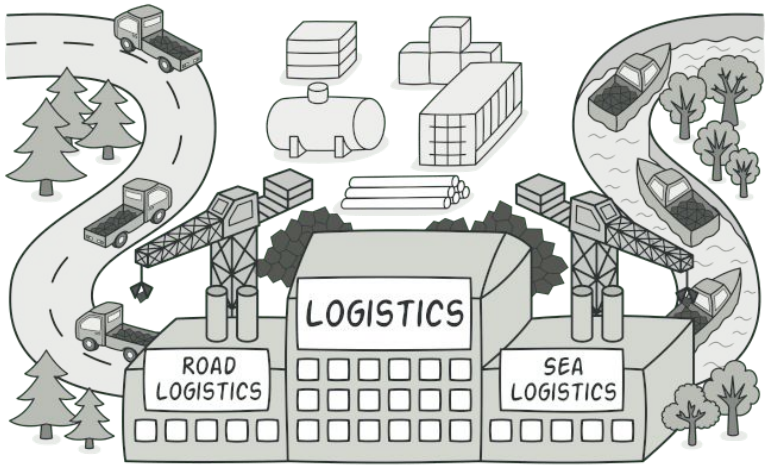
El proceso de construcción debe permitir diferentes representaciones para el objeto que se está construyendo.

Factory method

Parte de Patrones de creación.

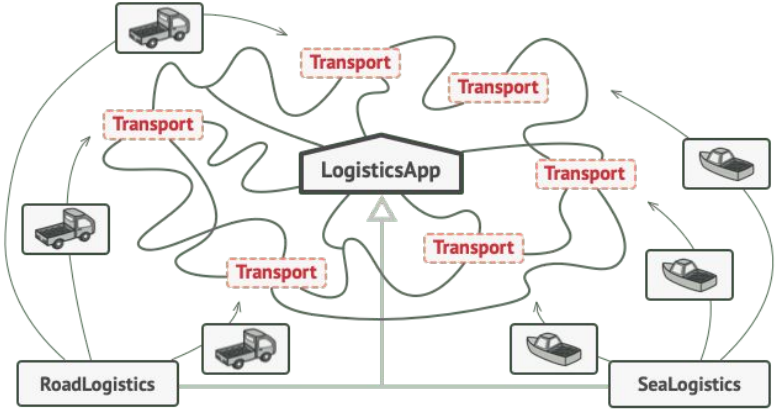
Define una interfaz para crear un objeto, pero deja a la subclase decidir cuál clase instanciar.

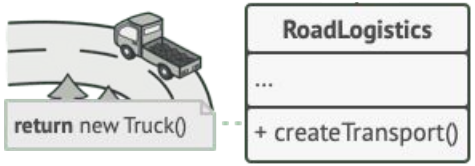
También conocido como Virtual Constructor.

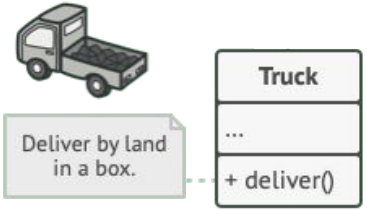


Problema

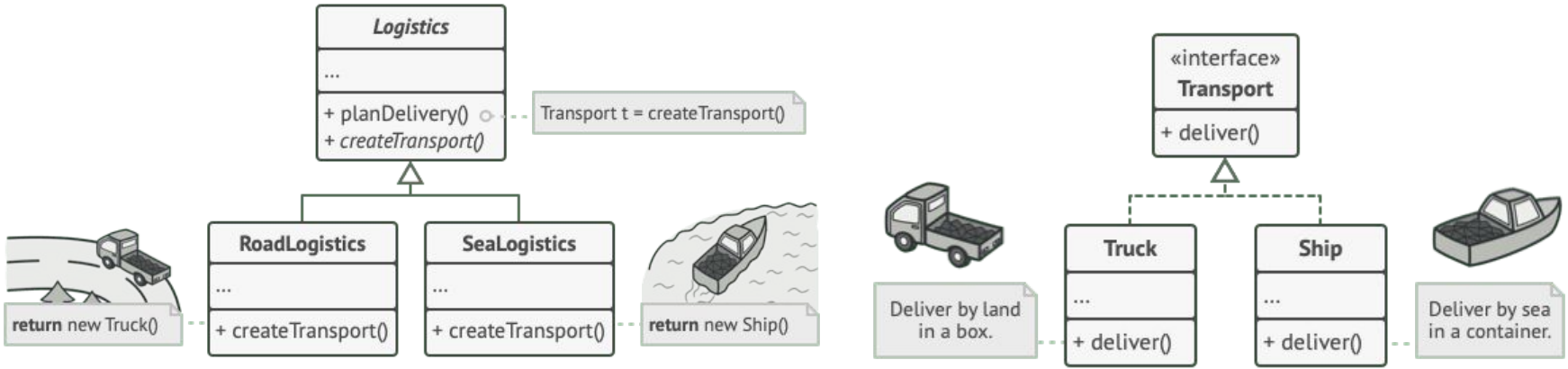

LogisticsApp


Transport
Transport
Transport
LogisticsApp
Transport
Transport
Transport
RoadLogistics
SeaLogistics


RoadLogistics
...
+ createTransport()
return new Truck()


Truck
...
+ deliver()
Deliver by land
in a box.

Solución


Logistics
...
+ planDelivery()
+ createTransport()
Transport t = createTransport()
RoadLogistics
...
+ createTransport()
SeaLogistics
...
+ createTransport()
return new Truck()
return new Ship()
«interface»
Transport
+ deliver()
Truck
...
+ deliver()
Ship
...
+ deliver()
Deliver by land in a box.
Deliver by sea in a container.

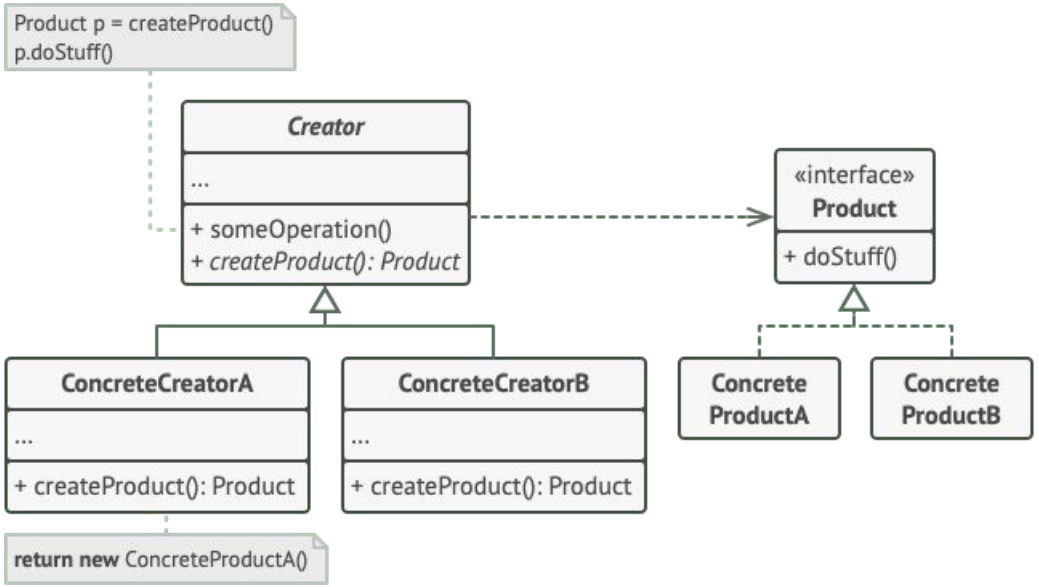
Estructura

Product: define la interfaz en común a todos los objetos que el factory produce.

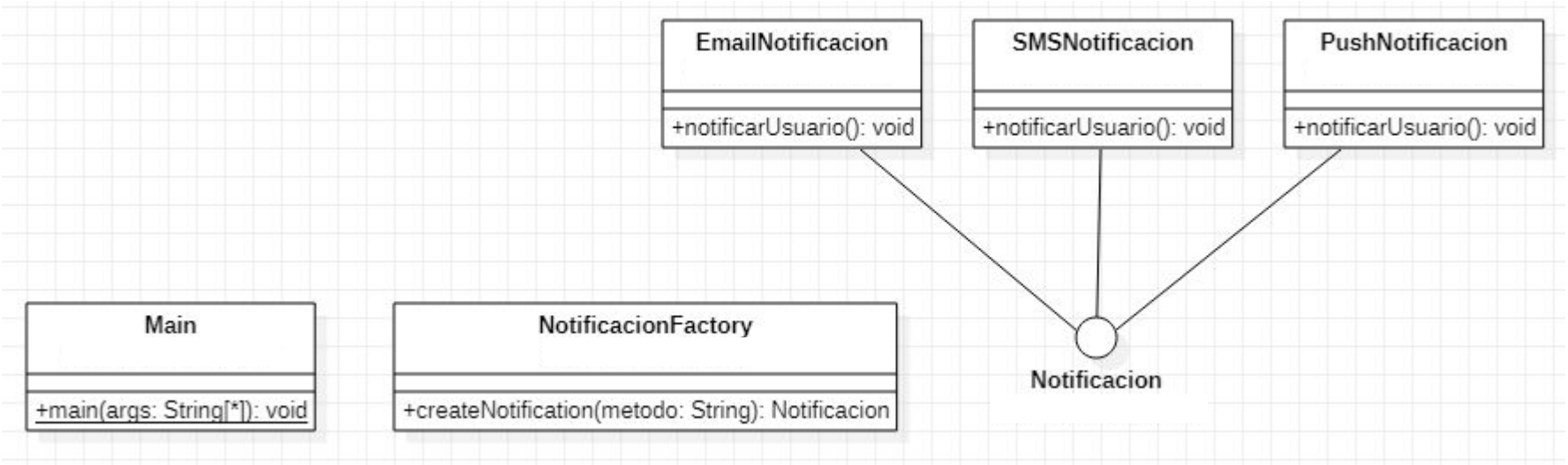
ConcreteProduct: implementaciones de la interfaz productos.

Creator: declara el factory method, que devuelve un objeto tipo Producto, también podría tener una implementación por defecto que devuelva un ConcreteProduct por defecto.

ConcreteCreator: sobreescribe el factory method para devolver una instancia de un ConcreteProduct



Ejemplo


EmailNotificacion
+notificarUsuario(): void
SMSNotificacion
+notificarUsuario(): void
PushNotificacion
+notificarUsuario(): void
Notificacion
Main
+main(args: String[*]): void
NotificacionFactory
+createNotification(metodo: String): Notificacion

Usar cuando:

Una clase no puede anticipar que clase de objetos debe ser creada.

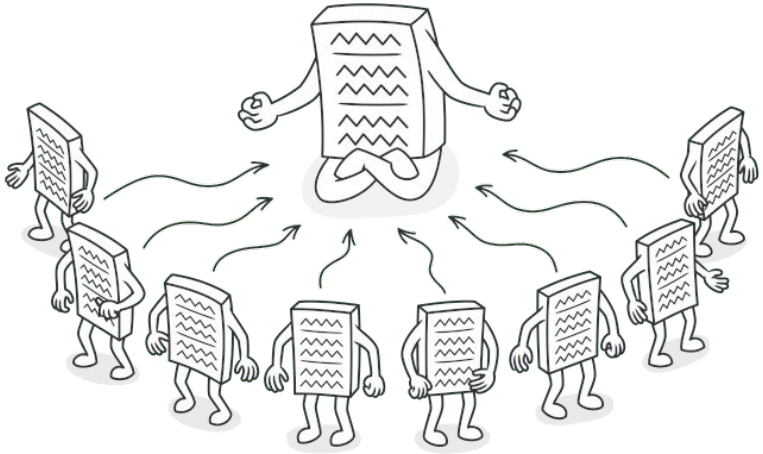
Una clase quiere que su subclase especifique el objeto a crear.

Las clases delegan la responsabilidad a una de sus muchas subclases, y es necesario ubicar cuál de todas esas subclases es la que puede dar respuesta a la necesidad actual.

Singleton

Parte de los patrones creacionales.

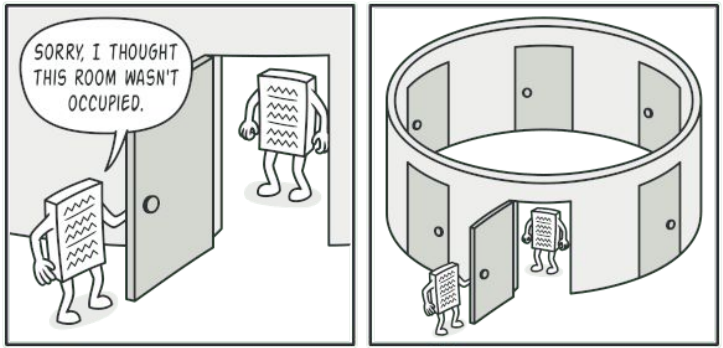
Nos asegura tener una única instancia de una clase y que esta sea accesible globalmente.



Problema

1- **Asegura una única instancia de la clase**: es necesario para controlar el acceso a recursos compartidos como base de datos o archivos.

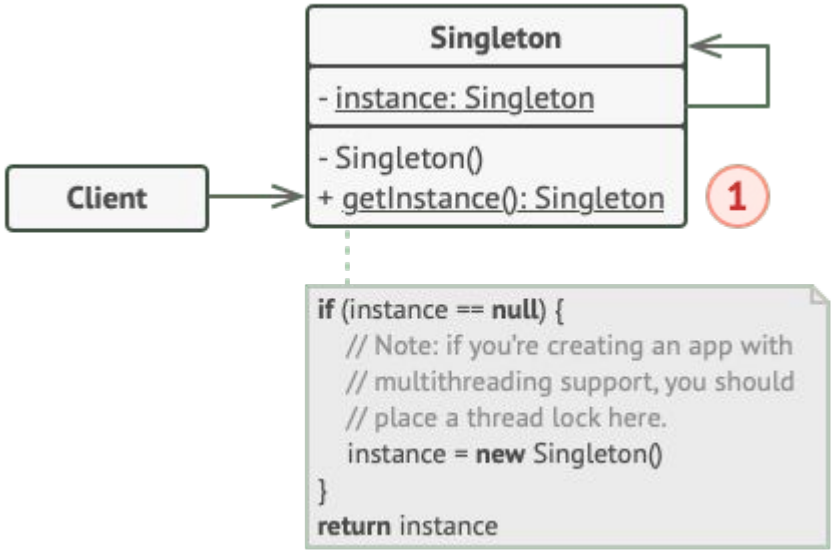
2- **Provee un punto de acceso a esa instancia**: las variables globales pueden llegar a ser inseguras si son modificadas en diferentes momentos de la ejecución. Además de poder acceder también se asegura de que no sea modificada la instancia.



Solución

1- Se declara el método static getInstance() que devuelve una instancia de su misma clase.

El constructor de la clase debe ser privado, el getInstance() debe ser la única forma de acceder.



Usar cuando:

Debe existir solo una instancia de una clase y debe ser accesible a clientes desde un solo punto.

Temario

- Patrones de diseño
- Creational patterns
  - Builder
  - Factory method
  - Singleton
- Structural patterns
  - Composite
  - Decorator
  - Facade
- Behavioral patterns
  - Observer
  - State
  - Strategy

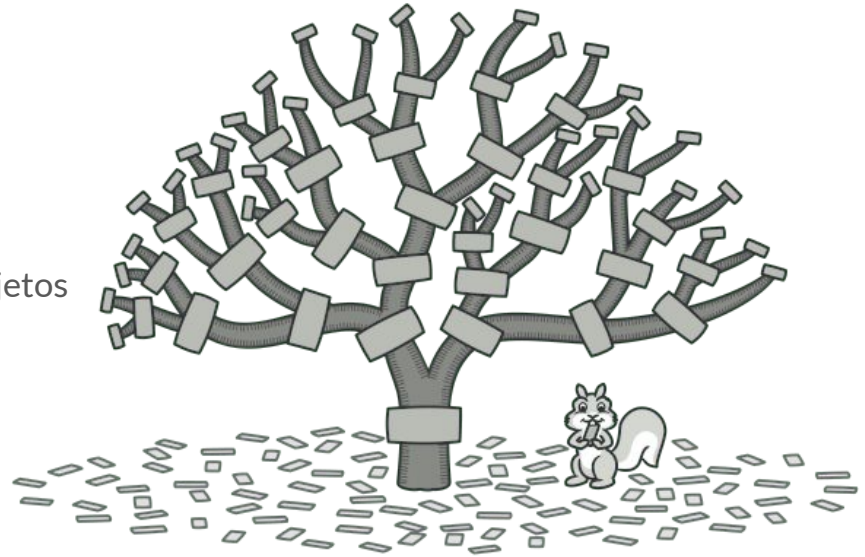
Structural patterns

Estos patrones de diseño tienen que ver con la composición de clases y objetos. Los conceptos de herencia se utilizan para componer interfaces y definir formas de hacer una composición de objetos para obtener nuevas funcionalidades.

Composite

Parte de Patrones Estructurales.

Nos permite tener una estructura de árbol de objetos compuestos y tratar a esta estructura como si fueran objetos individuales.

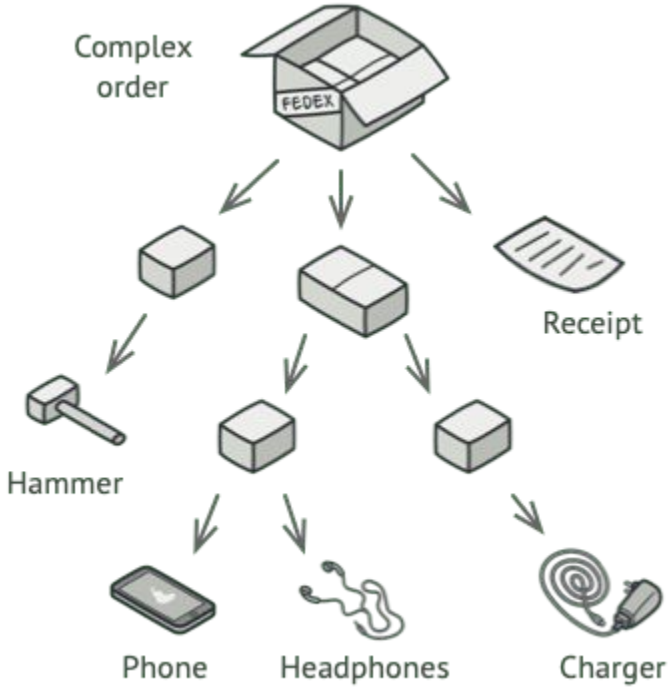


Problema

Tiene sentido sólo cuando el modelo central de la aplicación puede ser representado como un árbol.

Por ejemplo: Una orden de compra con varios productos dentro y que al mismo tiempo puede contener varias cajas con productos dentro y así...

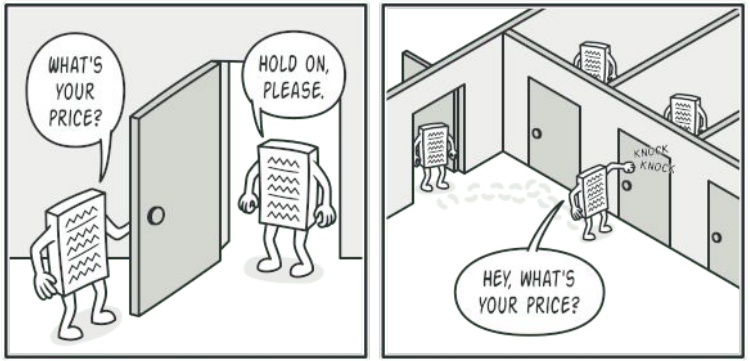
Para hacer una operación sobre cada uno de los elementos que puede contener (calcular el precio) sería necesario iterar y para esto, estar al tanto de niveles de anidación, el tipo de clases que se puede encontrar dentro y parámetros a consultar, puede llegar a ser complejo.



Solución

El patrón nos permitirá trabajar con los productos y cajas mediante una interfaz que declara los métodos comunes por ejemplo el cálculo del precio total.

Para un producto nos devuelve el precio, para una caja la sumatoria de los productos dentro.



El beneficio es que uno se puede abstraer de que tan compleja puede ser la estructura creada, uno pide lo especificado en la interfaz y los objetos mismos pasan el pedido en el árbol.

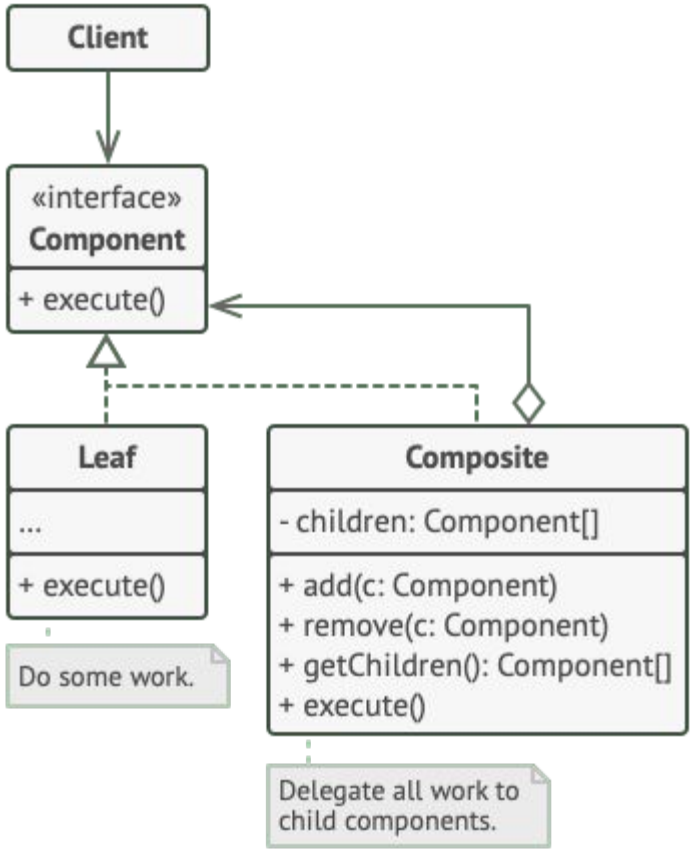
Estructura

1- **Component** interfaz que define las operaciones comunes a los objetos simples y complejos del árbol.

2- **Leaf** elemento básico de un árbol que no puede delegar el trabajo.

3- **Composite** (o container) contiene los subelementos (otros composite o leaf) siempre trabaja mediante la interfaz. Cuando recibe un pedido lo delega a los subelementos.

4- **Client** Trabaja con los componentes mediante la interfaz.



Usar cuando:

Cuando se quiere representar parte o toda una jerarquía de objetos.

Se quiere que los clientes puedan ignorar las diferencias entre la composición de objetos o de los objetos individuales. El cliente va a tratar a todos los objetos como una estructura compuesta uniforme.

Decorator

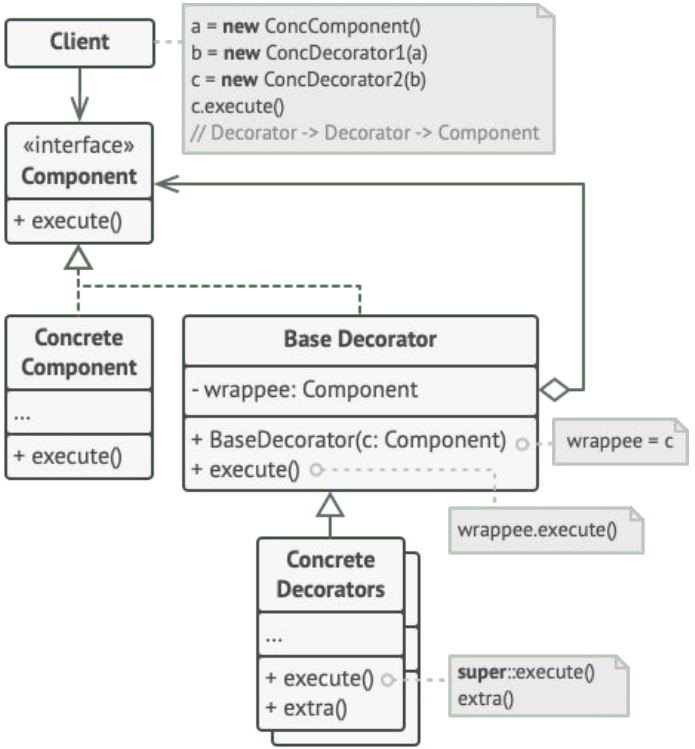
Parte de patrones estructurales.

Añade responsabilidades adicionales a un objeto dinámicamente. Decorator provee una alternativa flexible a hacer subclases para extender funcionalidades.

también conocido como Wrapper

Estructura

- **Component**: define interfaz para objetos que pueden tener responsabilidades nuevas.
- **ConcreteComponent**: define un objeto al cual se le puede agregar nuevas responsabilidades
- **Base Decorator**: mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la interfaz Componente, delega todo a Componente.
- **ConcreteDecorator**: añade las responsabilidades al componente



Usar cuando:

Se necesita añadir responsabilidades a objetos individualmente de forma dinámica y transparente (sin afectar a otros objetos).

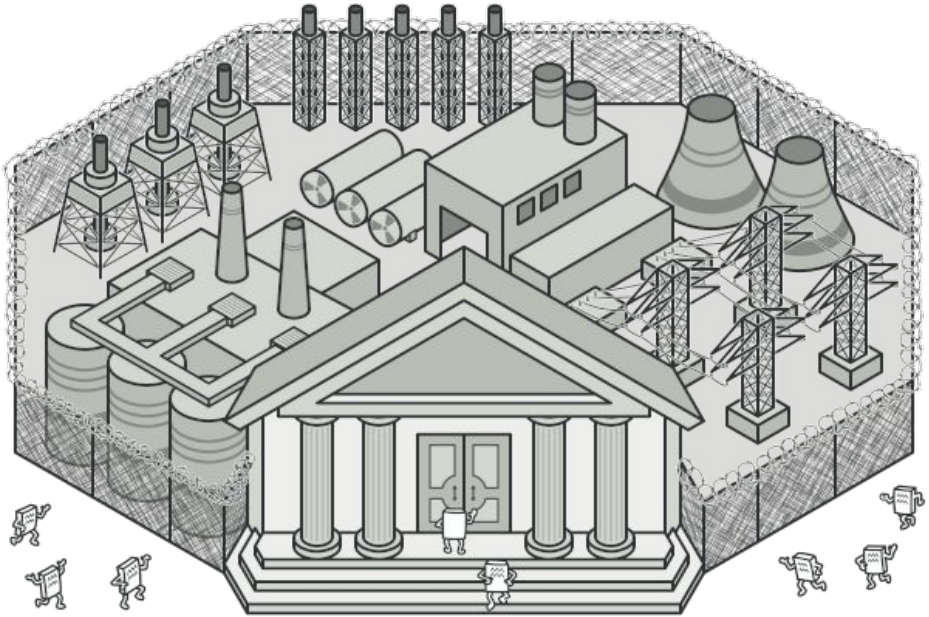
Para responsabilidades que pueden ser quitadas.

Cuando la extensión con subclases es impráctica, pj. exceso de subclases, no se tiene visibilidad del padre, etc..

Facade

Parte de los patrones estructurales.

Provee una interfaz de acceso simple a una librería, framework, o cualquier conjunto complejo de clases.



Problema

Hacer que tu código trabaje con un conjunto grande objetos que pertenecen a una librería, framework o API compleja:

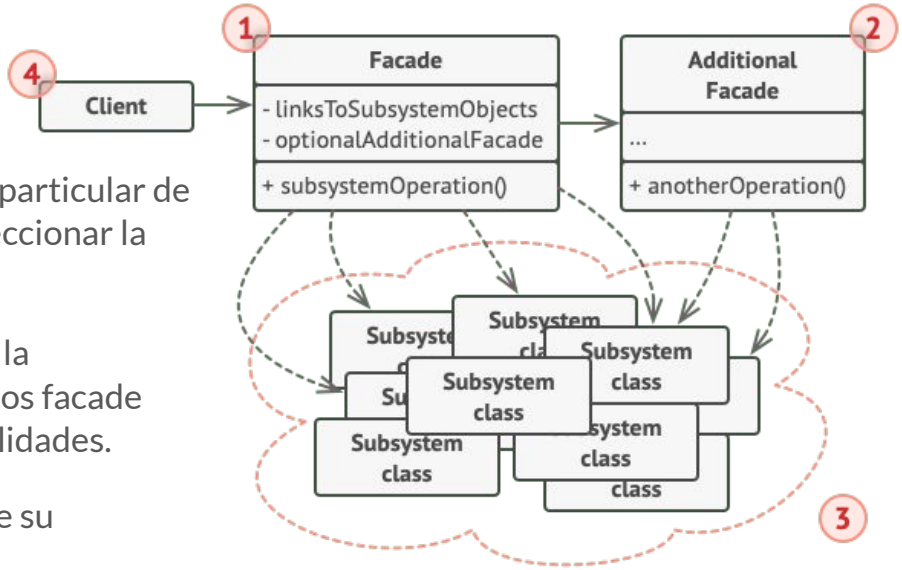
Inicializar muchos objetos.

Mantener las dependencias.

Ejecutar métodos en el orden correcto. Etc..

La lógica de tu programa termina muy acoplada a detalles de implementación de clases de terceros, haciendo que sea difícil de mantener y comprender.

Solución



1- **Facade** provee un acceso conveniente a una parte en particular de las funcionalidades del subsistema. Conoce a dónde direccionar la petición del cliente.

2- **Additional Facade** puede llegar a estar para asegurar la responsabilidad única del Facade original. Se añaden otros facade para encapsular coherentemente los accesos a funcionalidades.

3- **Subsystem class** de lo que tratamos de abstraernos de su implementación y simplificar su acceso con un facade.

4- **Client** que hace uso del facade a las funcionalidades de los subsistemas complejos.

Usar cuando:

Se quiere proveer una interfaz simple a un subsistema complejo.

Se tienen muchas dependencias entre clientes y las clases que implementan abstracciones. Para desacoplar el subsistema de los clientes, promoviendo así la independencia y portabilidad.

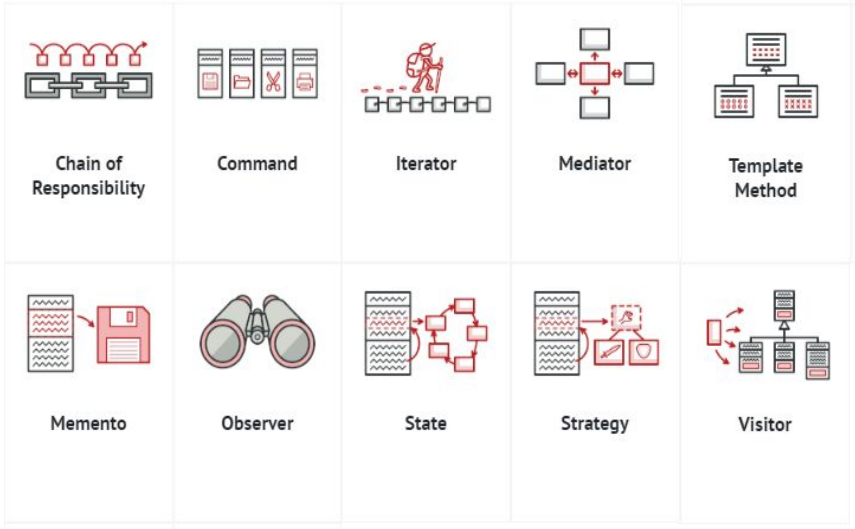
Se quiere implementar capas en el subsistema, cada Facade sería la puerta de entrada a un nivel del subsistema.

Temario

- Patrones de diseño
- Creational patterns
  - Builder
  - Factory method
  - Singleton
- Structural patterns
  - Composite
  - Decorator
  - Facade
- Behavioral patterns
  - Observer
  - State
  - Strategy

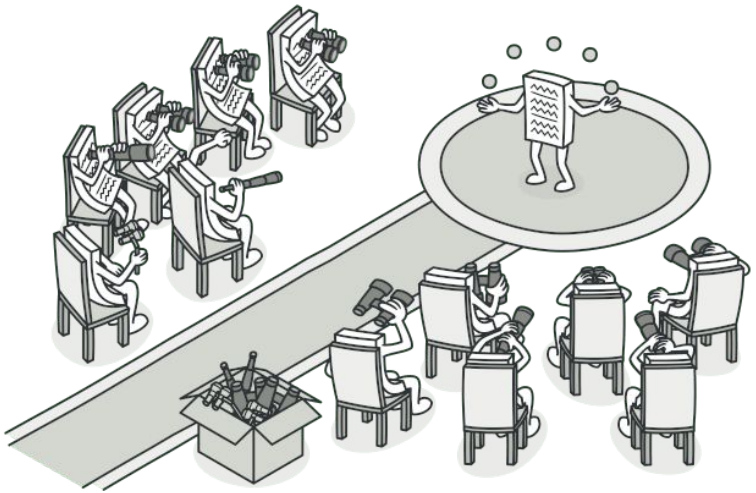
Behavioral patterns

Relacionados específicamente con la comunicación entre objetos, con los algoritmos y la asignación de responsabilidades entre los objetos.



Observer

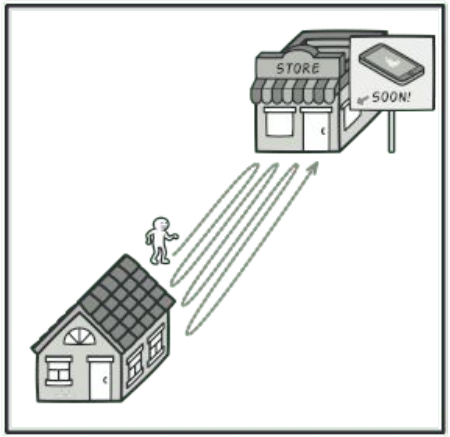
Permite definir un mecanismo de suscripción para notificar a múltiples objetos acerca de los eventos que suceden al objeto observado.

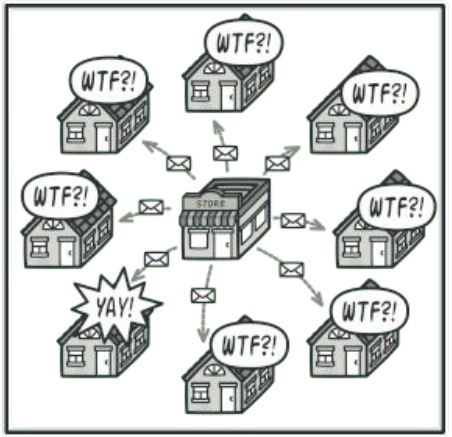


Problema

Un cliente quiere saber cuándo va a estar disponible el último teléfono que sale al mercado, para esto va al local todos los días.

El local avisa a todos que tiene disponibilidad del nuevo teléfono.



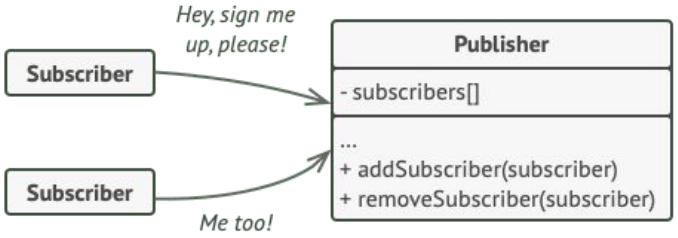


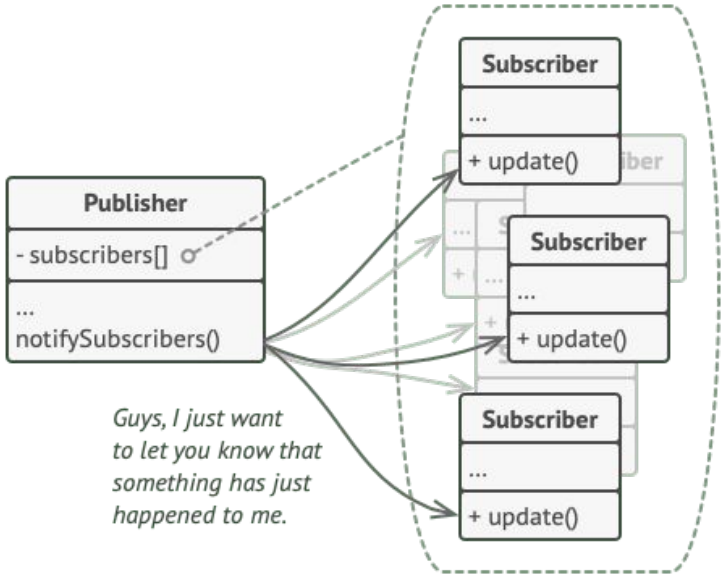
En una opción el cliente pierde tiempo y en la otra el local recursos al notificar a quien no está interesado.

Solución

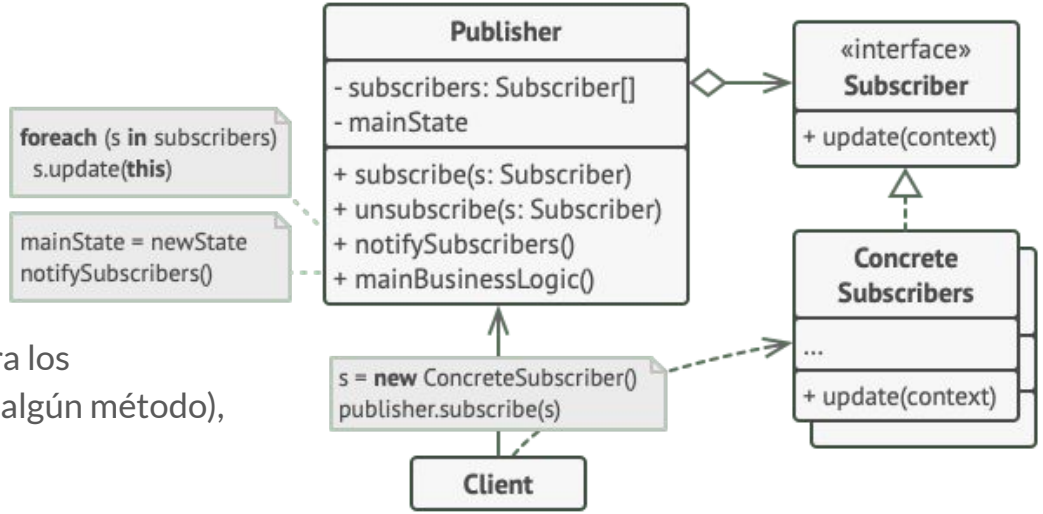
Los objetos se suscriben a un publicador.

Cuando algo importante sucede al publicador, este notifica a todos los suscriptores.





Estructura



1- **Publisher** Maneja los eventos de interés para los suscriptores (cambio de estado o ejecución de algún método), los publica, permite suscripción y desuscribir.

2- **Suscriber** es una interfaz que declara el método para notificar a los suscriptores.

3- **ConcreteSubscriber** hacen alguna acción en base a lo que se notifica, todos las clases deben implementar la interfaz así el publisher no se ve acoplado a las clases concretas.

4- El **Client** se encarga de crear los Publisher y Suscribers de forma separada y de hacer las asociaciones entre ellos y los updates.

Usar cuando:

Una abstracción tiene dos aspectos, uno dependiente de otro. Encapsular estos objetos de forma separada permite variar y reusarlos de manera independiente.

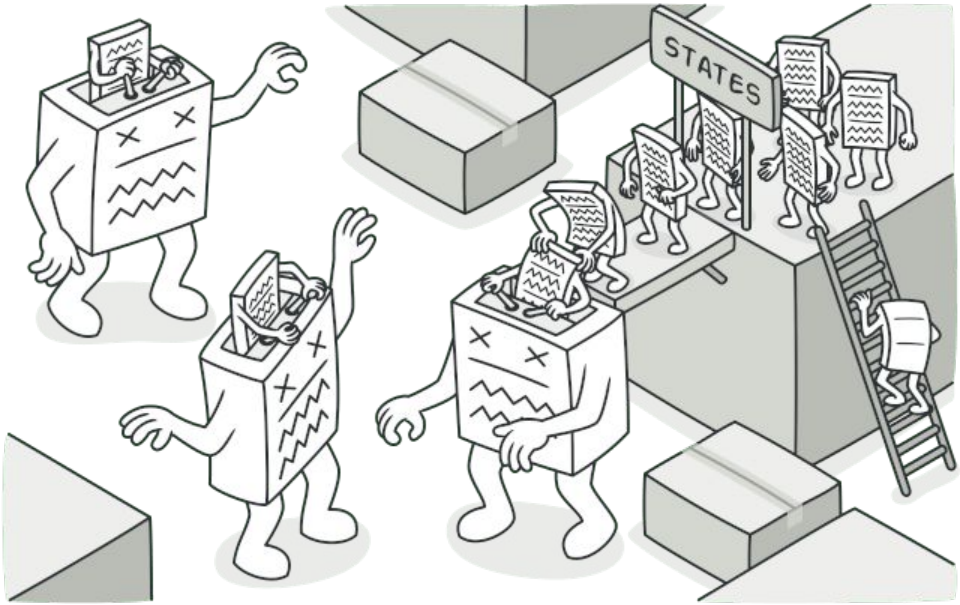
Un cambio en un objeto requiere cambios en otros y no se sabe cuántos objetos deben ser cambiados.

Un objeto debe notificar a otros sin hacer suposiciones sobre qué objetos se está tratando. En otras palabras, para no tener objetos altamente acoplados.

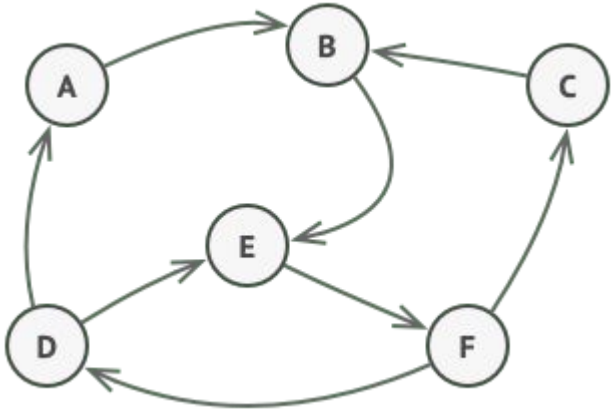
State

Parte de los patrones de comportamiento.

Permite alterar el comportamiento cuando su estado interno cambia.



Problema



Relacionado al concepto de una máquina de estados finitos:
La idea principal es que en un momento específico, hay un estado finito de estados en el que el programa puede estar, el programa se comporta diferente y este puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo del estado actual, el programa puede o no variar a otros estados. Estas reglas de cambio se llaman transiciones, también finitas y predeterminadas.

Por ejemplo un doc en google drive puede estar en un estado de “edición” “sugerencias” y “visualización”, cada una se comporta distinto.

Si todas las variantes de acciones pasarán por una sola clase, esta tendría muchas estructuras if y switch en base al estado actual.

ejemplo: sistema pago o premium

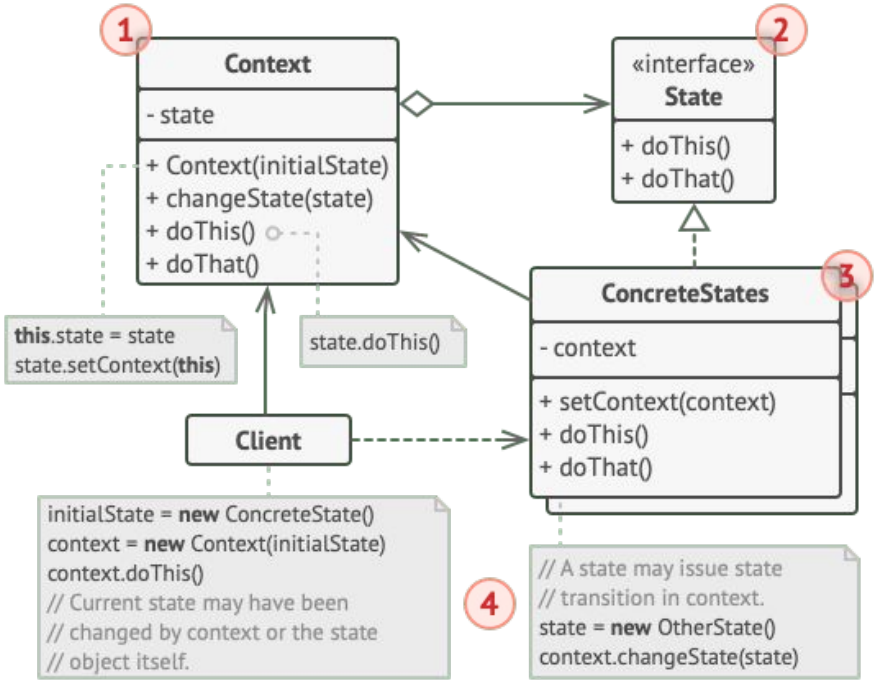
Solución

1 - **Context** almacena una referencia a un Concrete State y le delega todo el trabajo específico de estado. Context además expone el set del state para poder cambiar de estado desde los ConcreteState.

2- **State** interfaz que declara los métodos específicos, tienen que tener sentido para todos los ConcreteStates.

3- **Concrete States** provee la implementación de los métodos específicos. Se puede usar una estructura de herencia para evitar duplicar código.

4- Context o los ConcreteState pueden setear estado para transicionar a otro.



Usar cuando:

El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución.

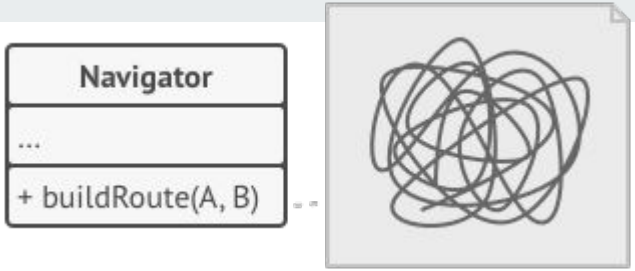
Una operación tiene muchas sentencias condicionales que dependen del estado del objeto. El estado suele estar representado por una o varias constantes.

Strategy

Parte de los patrones de comportamiento.

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Strategy deja al algoritmo variar independientemente de los “clientes” que los usen.

también conocido como Policy



Problema

Supongamos una aplicación que crea un mapa con intereses turísticos y las formas de ir de un lado a otro. Inicialmente trazaba rutas para ir en auto pero con el tiempo se agregaron nuevas formas de transporte:

a pie, transporte público, bicicleta, etc..

El código se vuelve un desastre (Bloated code)

Solución

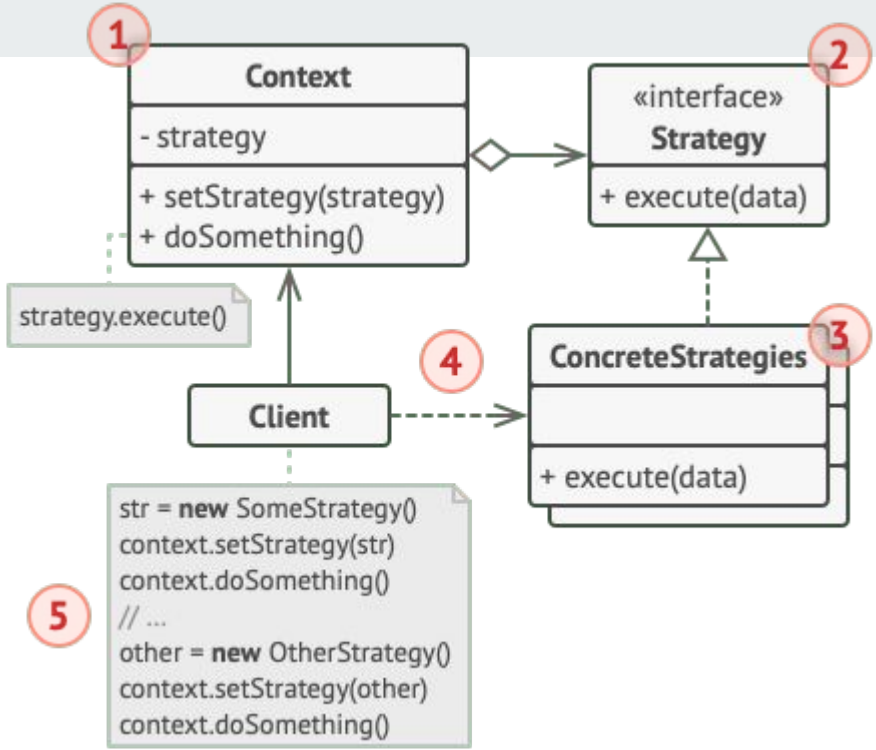
1- **Context:** mantiene referencia a la estrategia y la utiliza.

2- **Strategy:** Interfaz, declara que todas las estrategias tienen que poder ejecutar una estrategia.

3- **ConcreteStrategies:** le dan cuerpo.

4- Contexto llama a la estrategia, y no sabe con qué estrategia se esta ejecutando.

5- el **cliente** instancia una estrategia y se la da al contexto, el contexto expone un set de la estrategia para que esta pueda ser modificada.



Usar cuando:

Muchas clases relacionadas sólo difieren en su comportamiento, strategy provee una forma de configurar una clase com muchos comportamientos.

Necesitas diferentes variantes de un algoritmo. (pj cuando varía con el tiempo)

Un algoritmo usa información que el cliente no debería saber. Para evitar exponer estructuras de datos complejas y específicas del algoritmo.

Una clase define muchos comportamientos y aparecen como múltiples condicionales. Si esto aparece cambiar los múltiples if por clases que implementan cada una de las estrategias.