diseño de sistemas

utn frlp

U3 – Patrones de Diseño – Parte 1

Temario

- Diseño Orientado a Objetos (DOO)
- Revisión de Principios: **SOLID**
- ¿Qué es un patrón?
- Ventajas e inconvenientes.
- Una clasificación de patrones de diseño.
- Presentación de patrones de diseño.
- Ejemplos de aplicación de patrones.

utn frlp ds 

Revisión de Principios…

Inicial	Signifi-cado	Concepto
S	SRP	Single Responsability Principle. Un objeto debería tener una única responsabilidad
O	OCP	Abierto/Cerrado: las entidades de software deben estar abiertas a la extensión, pero cerradas para su modificación.
L	LSP	Sustitución de Liskov: instancias de subtipos reemplazan al supertipo sin alterar el correcto funcionamiento del programa.
I	ISP	Segregacion de interfaces (particionamiento): es preferible muchas interfaces cliente específicas a una de propósito general.
D	DIP	Inversión de dependencia: Es preferible depender de abstracciones, no de concreciones.

utn frlp ds 

Patrón

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces sin hacer lo mismo dos veces"

Cristopher Alexander

Nombre descriptivo abstracción

Problema cuándo aplicarlo, contexto?

Solución qué elementos la componen, relaciones

Consecuencias resultados, ventajas y desventajas

utn frlp ds 

Patrón

Un patrón es una **solución probada** que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de sistemas software.

No son bibliotecas de clases, sino un “esqueleto” básico que cada desarrollador adapta a las peculiaridades de su aplicación.

Los patrones se describen en **forma textual**, acompañados de diagramas (habitualmente de clases e interacción) y pseudocódigo.

Se deben **distinguir** de los estilos arquitectónicos y, en particular, de los frameworks.

utn frlp ds 

Ventajas y Desventajas

Incorporar los patrones de diseño como una herramienta más para el desarrollo de software de **calidad**.

Adquirir un nuevo **vocabulario** específico de patrones para comunicar soluciones de diseño.

Describir las **características** básicas de los patrones de diseño.

Entender las ventajas de los patrones en términos de unidades **reusables**.

Adquirir criterios para **aplicar efectivamente** patrones y para seleccionar el **patrón adecuado** a cada situación.

Facilitan la **reutilización de código**, pero un buen diseño es la clave para una reutilización efectiva.

utn frlp ds 

Ventajas e Inconvenientes

Son soluciones concretas:
Un catálogo de patrones es un conjunto de recetas de diseño.
Aunque se pueden clasificar, cada patrón es independiente del resto.

Son soluciones técnicas:
Dada una determinada situación, los patrones indican cómo resolverla mediante un D.O.O. Existen patrones específicos para un lenguaje determinado, y otros de carácter más general.

Se aplican en situaciones muy comunes:
Proceden de la experiencia. Han demostrado su utilidad para resolver problemas que aparecen frecuentemente en el D.O.O.

utn frlp ds 

Ventajas e Inconvenientes

Son soluciones simples:

Indican cómo resolver un problema particular utilizando un pequeño número de clases relacionadas de forma determinada.
No indican cómo diseñar un sistema completo, sino sólo aspectos puntuales del mismo.

Facilitan la reutilización de las clases y del propio diseño:

Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables.
La propia estructura del patrón es reutilizada cada vez que se aplica.

utn frlp ds 

Ventajas e Inconvenientes

El uso de un patrón no se refleja claramente en el código:

A partir de la implementación es difícil determinar qué patrón de diseño se ha utilizado, ya que queda incrustado en el modelo. No es posible hacer ingeniería inversa.

Referencias a "self":

Muchos patrones utilizan la delegación de operaciones. Tener en cuenta el correcto uso identificando el ámbito.

Es difícil reutilizar la implementación de un patrón:

Las clases del patrón son roles genéricos, pero en la implementación aparecen clases concretas.

utn frlp ds 

Ventajas y Desventajas

Un diseñador experimentado producirá diseños **más simples**, robustos y generales; fácilmente adaptables a cambios. (Flexibilidad – Escalabilidad)

Los patrones de diseño pretenden **explotar** soluciones efectivas a determinados problemas. (Reuso)

Los patrones suponen cierta **sobrecarga** de trabajo a la hora de implementar:
Se usan más clases de las estrictamente necesarias.
A menudo un mensaje se resuelve mediante delegación de varios mensajes a otros objetos

La idea es **reutilizar** la experiencia de quienes ya se han encontrado con problemas similares y han encontrado una buena solución

Los patrones no se inventan, se descubren

utn frlp ds 

Una Clasificación

PATRONES DE DISEÑO		
Según su propósito		
CREACIONALES	**ESTRUCTURALES**	**DE COMPORTAMIENTO**
Singleton	Adapter	Template Method
Prototype	Bridge	Observer
Factory Method	Composite	State
Abstract Factory	Decorator	Strategy
Builder	Facade	Command
	Flyweight	Interpreter
	Proxy	Chain of Resposibility
		Iterator
		Memento
		Visitor
		Mediator

utn frlp ds

Descubriendo Patrones. Caso 1

Supongamos que queremos modelar un editor gráfico.
En particular nos interesa el manejo de las figuras.

Podemos tener las siguientes figuras (elementos):

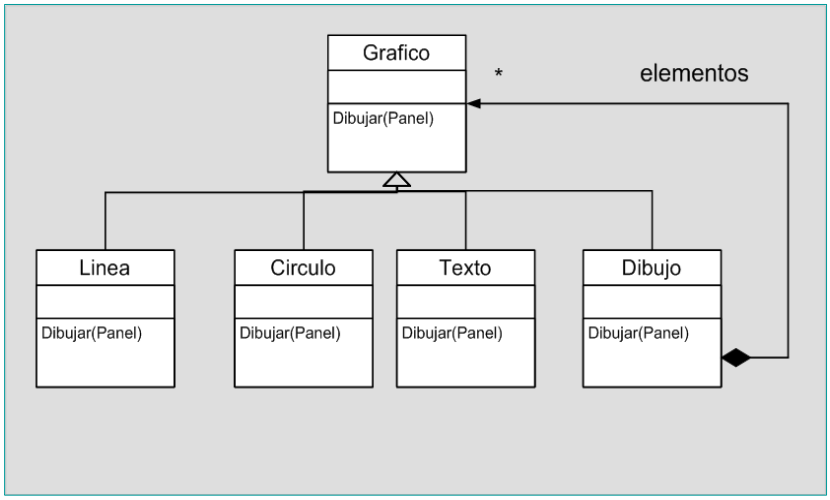
Línea **Círculo** **Texto** **Dibujo**

Un dibujo contiene uno o varios de los objetos en la lista.
Lo que interesa de las figuras es dibujarlas.
Un dibujo se dibuja dibujando las partes componentes

¿Cómo lo modelamos?

utn frlp ds

Una solución…



Nos permite construir estructuras de objetos en forma de árbol
Aplicar operaciones al elemento compuesto y los individuales
Ignoramos la estructura.

utn frlp ds

Otro caso

Supongamos que tenemos que diseñar un sistema que simula un filesystem de un sistema operativo. Se pueden tener archivos y carpetas.
En las carpetas se pueden tener archivos u otras carpetas.

Se quieren tener operaciones como:

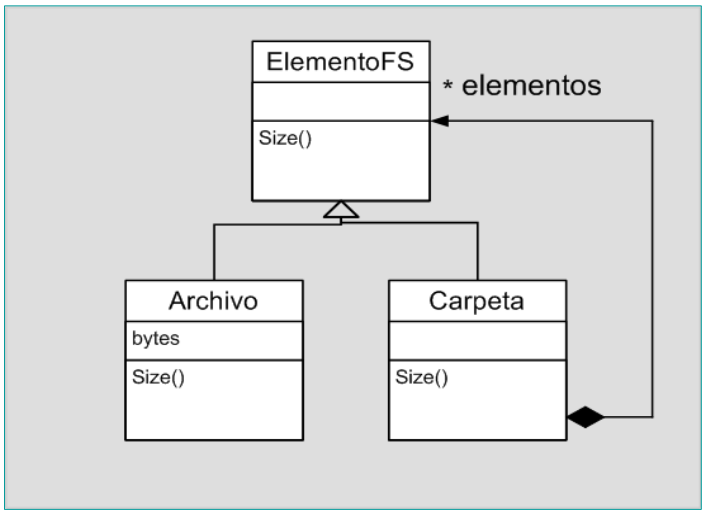
Tamaño

En un archivo es el tamaño del mismo
En una carpeta es el tamaño de los archivos que contiene.

¿Cómo lo modelamos?

utn frlp ds

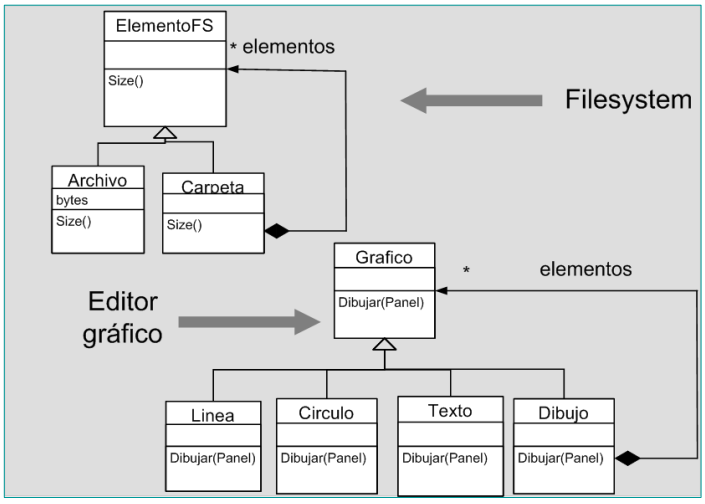
Una solución…



¿Cómo debería ser la operación size() en la interfaz?

utn frlp ds

Comparamos…



utn frlp ds

Una solución…

¿Tienen algo en común el problema/solución de los casos anteriores?
Recurrencia

¿Una vez diseñado el primer ejercicio, costó trabajo hacer el segundo?
Aprendizaje

¿Qué reutilicé? ¿Objetos? ¿El esquema de la solución?

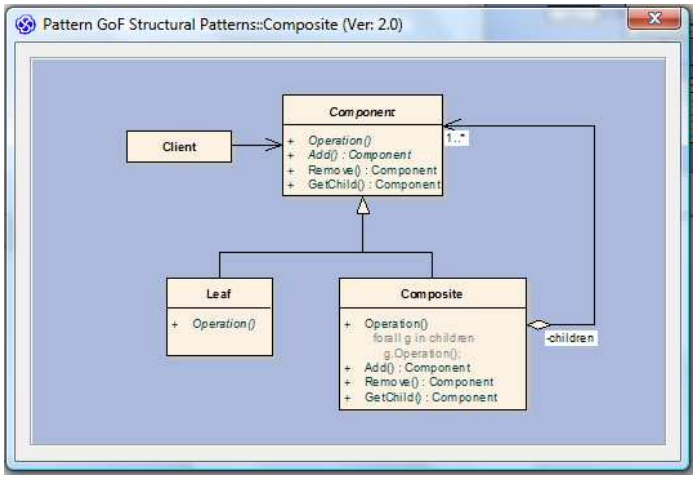
Tratamiento de un grupo o estructura compuesta del mismo modo (polimórficamente) que un objeto no compuesto (atómico)

Lo que aplicamos fue el patrón de diseño Composite.

Permite construir objetos de complejidad mayor mediante otros más sencillos de forma recursiva. Los clientes trabajan de igual manera.

utn frlp ds

Composite – Forma Genérica



Caso Práctico: un mozo debe conocer el menú, el cual puede ser un item de menú (Ej. papas fritas) o un plato combo por (papas fritas + milanesa, etc.)

utn frlp ds

Descubriendo Patrones. Caso 2

En una empresa existen dos tipos de empleados

Los empleados de **planta permanente**, que cobran la cantidad de horas trabajadas por $30, más monto por antigüedad y un monto fijo por salario familiar. Asimismo tienen un descuento por la obra social.

Los empleados de **planta temporaria**, que cobra la cantidad de horas trabajadas por $20, más un monto fijo salario familiar (no cobran antigüedad ni pagan obra social).

Se quiere calcular el sueldo de los empleados

¿Cómo lo modelamos?

utn frlp ds 

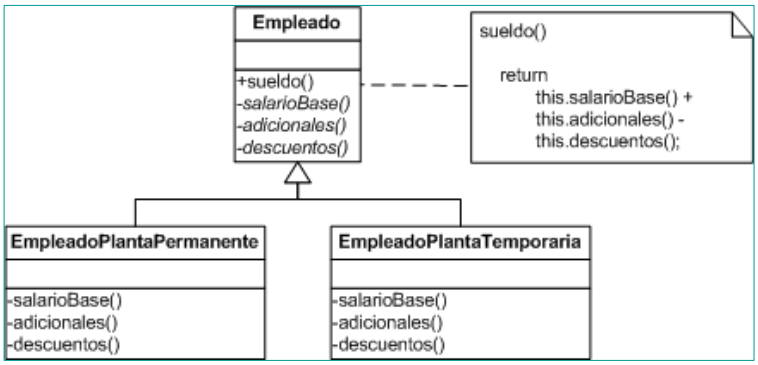
Descubriendo Patrones. Caso 2

Problema:

- En el caso que sea necesario definir el invariante de un algoritmo y permitir que las subclases modifiquen sus partes variables.
- En el caso que se encuentre comportamiento común en un conjunto de subclases que pueda ser refactorizado en la superclase.
- Para brindar puntos de extensión a la clase en forma controlada.

utn frlp ds 

Seguimos descubriendo…



utn frlp ds

Solución

Solución:

Definir una clase abstracta que implemente la estructura del algoritmo haciendo llamados a otros mensajes

Las subclases concretas deben definir los métodos de los pasos del algoritmo.

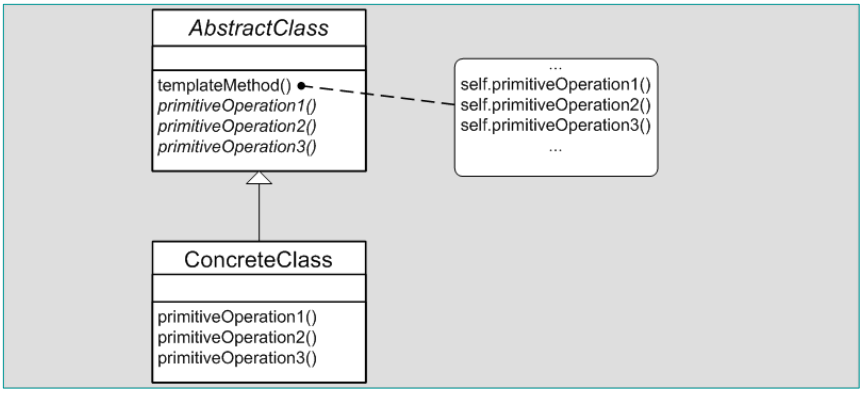
De esta forma el orden de los pasos queda fijo, pero cada clase puede realizar las variaciones de comportamiento necesarias en cada paso.

El comportamiento común se factoriza en la superclase y el específico (polimórfico) en las subclases.

Hemos aplicado el patrón de diseño Template Method.

utn frlp ds

Template Method



Técnica fundamental: factorizar en la superclase

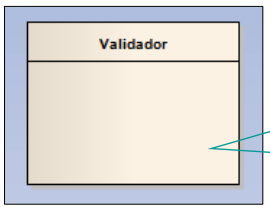
utn frlp ds

Seguimos descubriendo…

Caso:

Un negocio de venta con tarjeta de crédito tiene en su local existen varios puestos de facturación. Queremos incorporar al modelo de clases un sistema externo validador de tarjetas de crédito conectado a través de una línea dedicada. Cualquiera de los puestos de facturación envía mensajes al validador. Se requiere modelar la comunicación entre los objetos que demandan el servicio y el proveedor que es único.

¿Cómo evitamos que se creen varios validadores?




```
public class Validador {
    private void Validador(){
            }
// Al declarar el constructor privado solo la
clase puede instanciar esta clase.  Pero
cómo?  Debemos agregar un método de
clase que lo haga
}
```

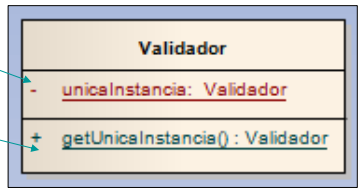
utn frlp ds

Analizamos un poco de código…

Solución: En la clase Validador definimos al constructor como privado.
Declaramos una variable estática para guardar la instancia única.
Definimos un método de clase que nos muestre el contenido de la variable estática (si no está creada la crea)

```
public class Validador {
        private static Validador unicaInstancia;

        private void Validador(){        (Constructor)
        }
        public static Validador getUnicaInstancia(){
           if (unicaInstancia == null ) {
                unicaInstancia = new Validador();
                }
                return unicaInstancia;
        }
}
```



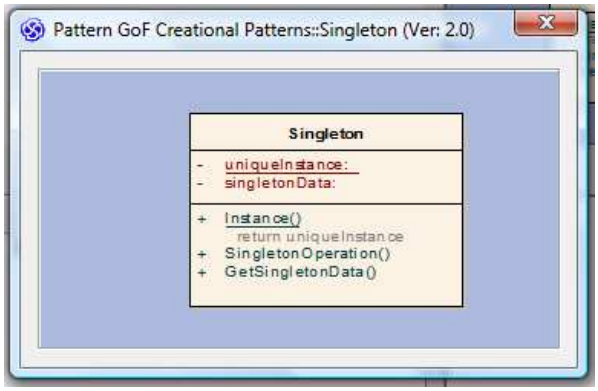
Resta que el objeto cliente que conoce al Validador le solicite la instancia a través del método de clase:

Validador.getUnicaInstancia()

Hemos aplicado el Patrón Singleton

utn frlp ds

Patrón Singleton



Forma genérica del patrón creacional.

utn frlp ds

Repasamos….

Hemos presentado:

Patrón Composite

Patrón Template Method

Patrón Singleton

Creacionales:
Tienen que ver con el proceso de creación de objetos.

Estructurales:
Tratan la composición de clases u objetos.

De comportamiento:
Muestran como las clases y objetos colaboran, interactúan y se reparten responsabilidades

Indique con una flecha la clasificación correspondiente de cada Patrón

utn frlp ds

Repasamos….

Hemos presentado:

Patrón Composite

Patrón Template Method

Patrón Singleton

Garantiza que una clase tenga una sola instancia y proporciona un punto de acceso a ella.

Combina objetos en estructura de árbol representando jerarquía partes-todo. Permite que los clientes a objetos individuales o compuestos uniformemente.

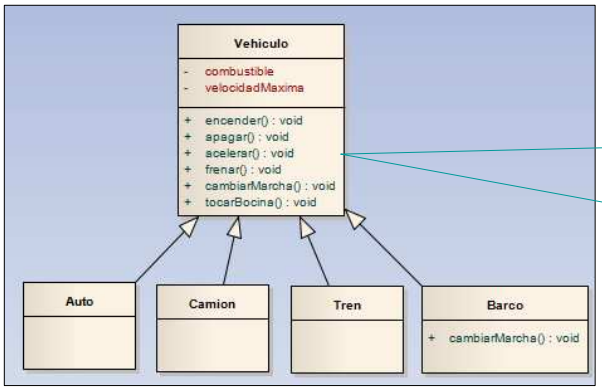
Define una operación esqueleto de un algoritmo, delegando en las subclases implementar las partes particulares sin cambiar su estructura.

Indique con una flecha la definición correspondiente a cada Patrón

utn frlp ds

Seguimos descubriendo

Caso:
Se desea realizar un modelo estático de distintos vehículos de transporte: pueden ser autos, camiones, trenes, barcos. Las operaciones que saben realizar son: encender, apagar, acelerar, frenar, cambiar de marcha, tocar bocina.



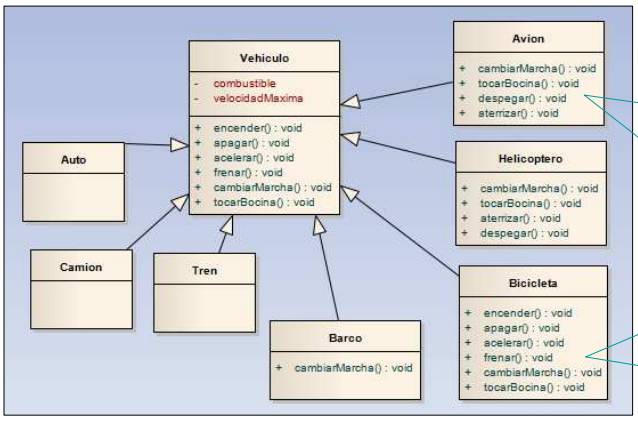
Establecemos que todas las operaciones son indistintas por lo tanto se codifican en la superclase.(Son concretas) Observamos que el barco no cambia la marcha, entonces Barco debe reimplementarlo con código (realiza nada). Aquellas operaciones que se realicen en forma particular se reimplementarán en las subclases.

Primera Versión del modelo

utn frlp ds

Continuación ….

Ahora agregar nuevas subclases avión, helicóptero, bicicleta. Los dos primeros también pueden despegar y aterrizar.



Tocar bocina no tiene sentido en esta subclase, habrá que reimplementar la operación para que no haga nada. Idem Helicoptero.

Habrá que reimplementar la operación tocarBocina para haga sonido de timbre. El resto de las operaciones no tienen sentido, nulificar el comportamiento.

Modificaciones en el comportamiento impactan en las subclases!!! Es una ventaja la herencia en este caso? El comportamiento es demasiado estático, no puede cambiarse en tiempo de ejecución.

utn frlp ds

Problema

- Existen muchos algoritmos para llevar a cabo una tarea. Mucha diversidad.
- No es aconsejable codificarlos todos en una clase y seleccionar cual utilizar por medio de sentencias condicionales.
- Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas y difíciles de mantener.
- Es necesario cambiar el algoritmo en forma dinámica.

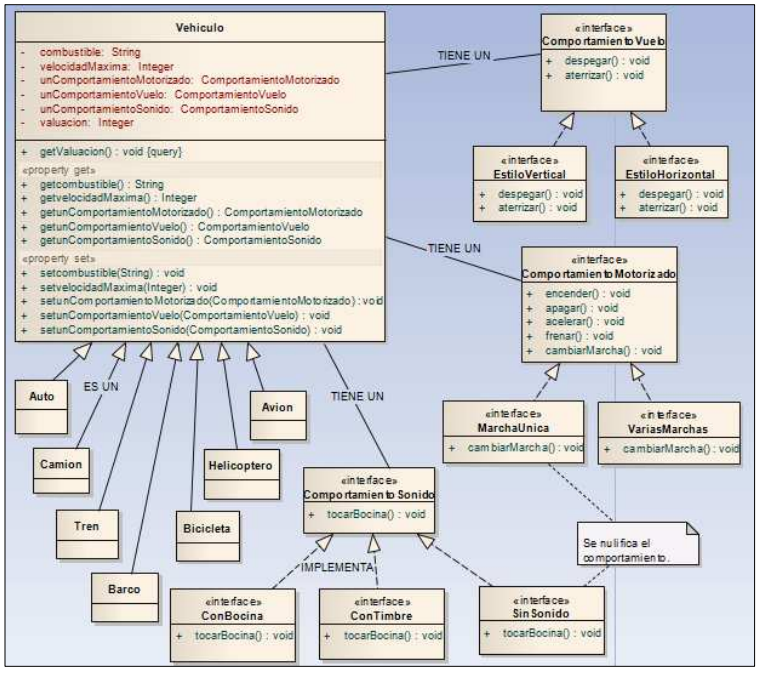
Solución

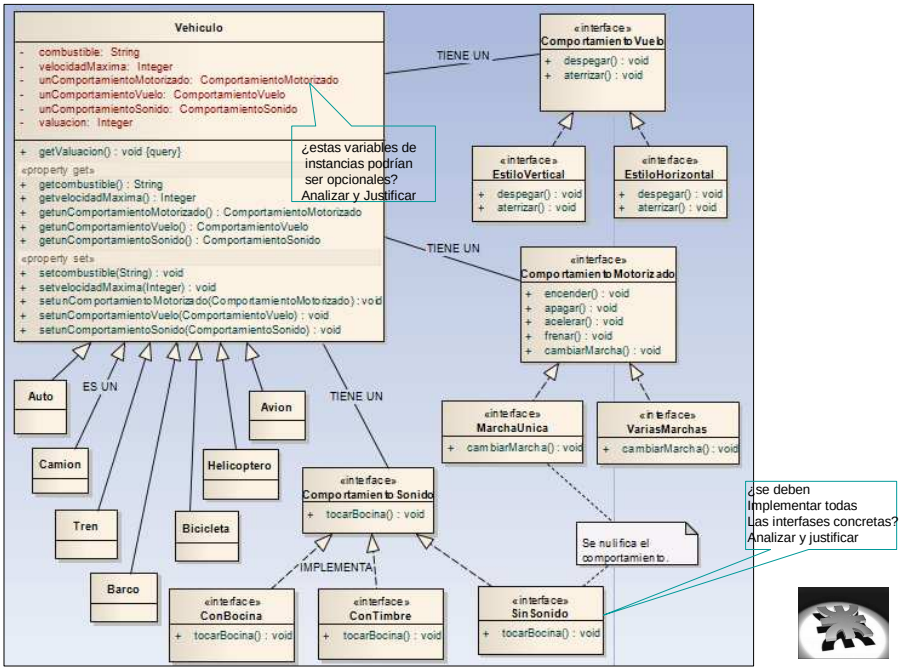
- Separar el comportamiento que varía de lo que no.
- Encapsular lo que varía para que no afecte al resto.
- Programar interfaces no implementar en clases.
- De esta manera una clase que implementa una interfaz no necesita conocer el detalle de implementación de comportamientos ajenos.

Vamos a aplicar el Patrón de Comportamiento Strategy

Primera Versión del modelo

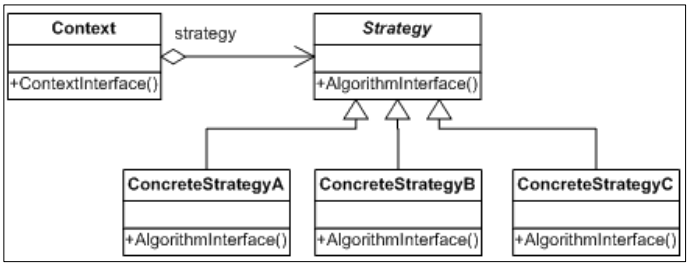
utn frlp ds





Patrón Strategy

Define una familia de algoritmos, encapsula a cada uno y los hace intercambiables.
Permite variar un algoritmo independientemente de quien lo usa (cliente).
Favorece la composición a la herencia estática. Utilizar **interfaces** separa el comportamiento de la estructura estática y permite cambiar comportamiento en tiempo de ejecución, posteriormente a la instanciación del objeto.
Flexibilidad ante nuevas subclases y nuevos comportamientos o variación en los mismos.



Extraído del Libro Gamma

utn frlp ds

Seguimos descubriendo…

Se trata de una Web que administra ofertas de productos de último momento.
Para recibir su catálogo un cliente se ha suscripto a la página.
Cada vez que se conforma un nuevo grupo de ofertas se envían por mail a cada cliente.
El cliente puede decidir no recibir más las noticias, y podrá cancelar la suscripción. Los clientes no conocen qué otras personas reciben las ofertas. ¿Cómo modelaría esta funcionalidad?

Problema:
Una clase cada vez que cambia su estado emite información (novedades) a otras clases dependientes que les interesa dicho cambio ya que afecta su propio estado el cual se actualiza automáticamente.

Solución:
Identificar roles: la clase independiente (Sujeto Observado), una interfaz Observador que la implementan las clases clientes interesadas. El sujeto va a incorporar nuevos objetos observadores a su lista y eliminar un observador de su lista que no desea ser notificado. Además el sujeto va a implementar una operación para notificar a todos sus clientes cada vez que se registre un cambio en su estado.

utn frlp ds 

Patrón Observer

Define una dependencia 1 a muchos entre objetos, cuando un objeto independiente cambia su estado todos los dependientes son notificados y actualizados automáticamente.

Consecuencias:

Brinda flexibilidad, ya que el objeto que dispara el cambio no sabe nada del protocolo de los objetos que son dependientes de él (bajo acoplamiento).
Bajo acoplamiento entre el observador y el sujeto (no depende del tipo de observer ni de la cantidad de los mismos).

Los interesados (observers) no están constantemente preguntando por cambios en el sujeto.

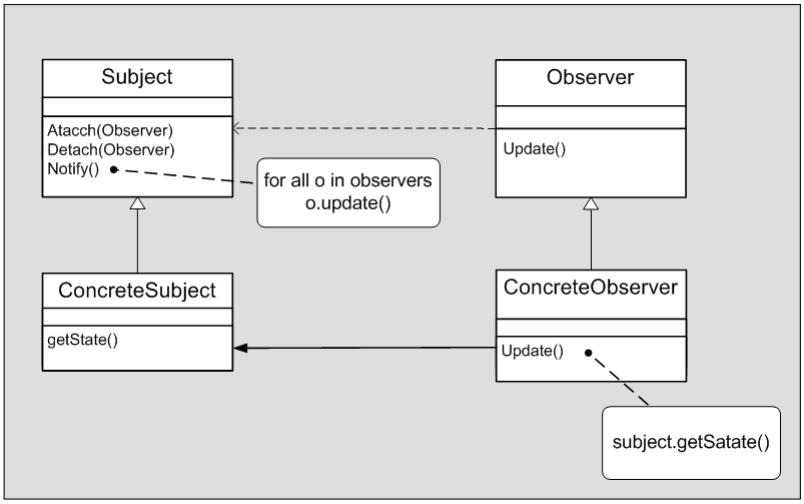
No es necesario especificar el receptor de la notificación.

Permite que diferentes observer se “subscriban” a diferentes características en el mismo subject.

Problemas de performance; un cambio en el sujeto puede disparar una cascada de actualizaciones.

utn frlp ds 

Patrón Observer



utn frlp ds

Patrón Decorator

Objetivo: Agregar comportamiento a un objeto dinámicamente y en forma transparente.

Problema: 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.

Solución: Diferenciar los objetos clientes y los que adicionan o decoran

Consecuencias: Permite mayor flexibilidad que la herencia.
Permite agregar funcionalidad incrementalmente

Patrón estructural

utn frlp ds

Patrón Decorator

Objetivo: Agregar comportamiento a un objeto dinámicamente y en forma transparente.

Problema: Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia.

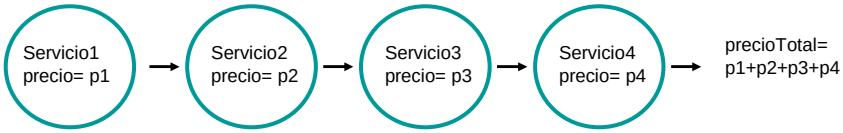
El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”.

El problema que tiene la herencia es que se decide estáticamente.

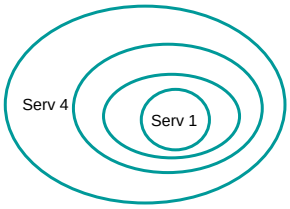
utn frlp ds

Patrón Decorator

Ejemplo: una empresa de turismo ofrece paquetes de viajes armados, no obstante los mismos pueden personalizarse con servicios adicionales. La cotización de su productos surge como la suma del paquete elegido más todos los adicionales. ¿Qué modelo propone y cómo se respondería el valor de cotización?



La idea es anidar los objetos realizando una composición:



Buscamos similitud con otro patrón ????

utn frlp ds

Patrón Decorator

Solución: Diferenciar los objetos clientes y los que adicionan o decoran, crear una subclase abstracta “el decorador” que estará compuesto de otro elemento de la superclase. Existe recursión y composición.

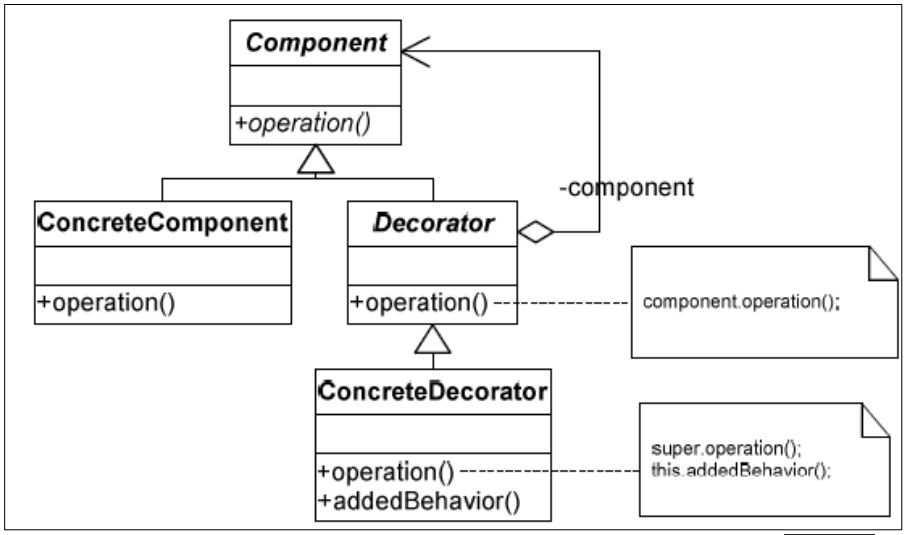
Consecuencias:

Permite mayor flexibilidad que la herencia.

Permite agregar funcionalidad incrementalmente.

utn frlp ds

Patrón Decorator



utn frlp ds

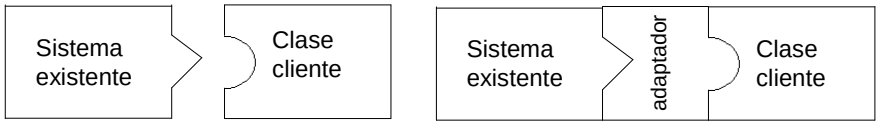
Patrón Adapter

Objetivo: Convertir la protocolo de una clase en otra, que es la que el objeto cliente espera. Crear una clase que se encargue de "transformar" los nombres de los mensajes.

Problema: Muchas veces, clases que fueron pensadas para ser reutilizadas no pueden aprovecharse porque su protocolo no es compatible con el protocolo específico del dominio de aplicación con el que se está trabajando.

Deseamos utilizar una clase existente, cuyo protocolo no encaja con la que necesitamos.

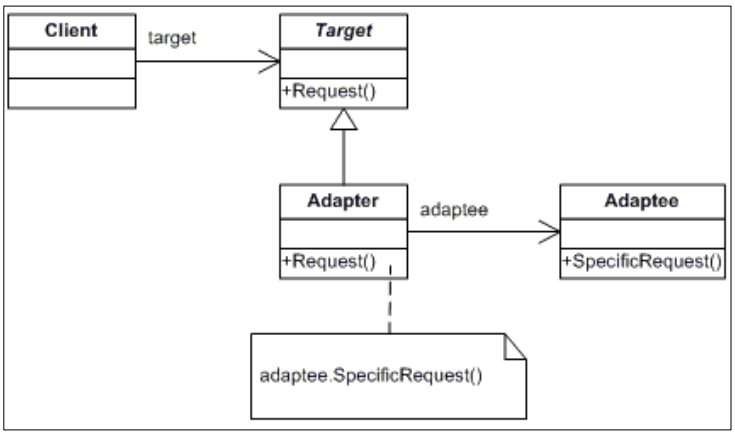
Deseamos crear una clase que puede llegar a cooperar con otros objetos cuyo protocolo no podemos predecir de antemano.



utn frlp ds

Patrón Adapter

El adaptador implementa la operación para transformar el protocolo y contiene una instancia de la clase a adaptar.



utn frlp ds

Patrón estructural

Patrón State

Objetivo: Modificar el comportamiento de un objeto cuando su estado interno se modifica. Externamente parecería que la clase del objeto ha cambiado.

Problema: Muchas veces el comportamiento de un objeto depende del estado en el que se encuentre.

En la programación procedural se suelen utilizar enumerativos y sentencias condicionales. Pero el código no escala. En objetos tenemos delegación y polimorfismo.

Solución: Desacoplar el estado interno del objeto en una jerarquía de clases.

Cada clase de la jerarquía representa un estado en el que puede estar el objeto.

Todos los mensajes del objeto que dependan de su estado interno son delegados a las clases concretas de la jerarquía (polimorfismo).

Patrón de comportamiento

utn frlp ds 

Analizamos un caso…

Queremos modelar un pedido de producto. Los posibles estados de un producto son: “en venta”, “reservado”, “vendido”, “en envío”, “entregado”, “rechazado”.

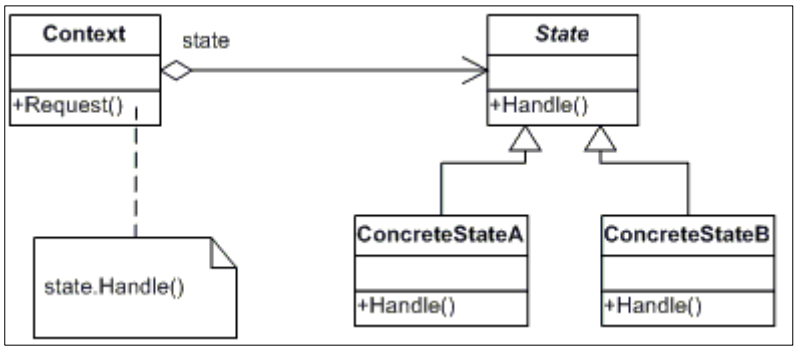
Creamos grupalmente la dinámica entre estados….

utn frlp ds 

Patrón State

Consecuencias:

Desacopla el comportamiento del objeto.
Localiza el comportamiento relacionado con cada estado.
Las transiciones entre estados son explícitas.



utn frlp ds

Repasamos….

Hemos presentado:

Patrón Decorator

Patrón Adapter

Patrón State

Creacionales:
Tienen que ver con el proceso de creación de objetos.

Estructurales:
Tratan la composición de clases u objetos.

De comportamiento:
Muestran como las clases y objetos colaboran, interactúan y se reparten responsabilidades

Indique con una flecha la clasificación correspondiente de cada Patrón

utn frlp ds

Repasamos….

Hemos presentado:

Patrón Decorator

Patrón Adapter

Patrón State

Permite la colaboración entre dos clases con protocolos incompatibles

Especifica el comportamiento de un objeto dependiendo del estado del mismo.

Enriquece el comportamiento de un Objeto adicionándole más responsabilidad

Indique con una flecha la definición correspondiente a cada Patrón

utn frlp ds 

PATRONES DE DISEÑO		
Según su propósito		
CREACIONALES	ESTRUCTURALES	DE COMPORTAMIENTO
Singleton	Adapter	Template Method
Prototype	Bridge	Observer
Factory Method	Composite	State
Abstract Factory	Decorator	Strategy
Builder	Facade	Command
	Flyweight	Interpreter
	Proxy	Chain of Resposibility
		Iterator
		Memento
		Visitor
		Mediator

Bibliografía:

- Patrones de Diseño, Autores: Gamma, Helm, Johnson, Vlissides (Addison Wesley)
- Design Patterns, Autores: Freeman & Freeman (Head First - Oreilly)
- UML y Patrones, Craig Larman (Prentice Hall)