Clase 9

Patrones de diseño

Patrones de diseño

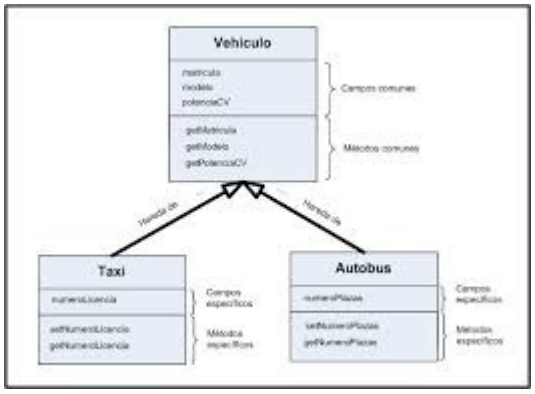
Programación orientada a objetos

Hacer software no es fácil

Diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos reutilizable es todavía más difícil.

Chapter 1: Introduction. Design Patterns, *The Gang of Four*

...y un software capaz de evolucionar tiene que ser reutilizable (al menos para las versiones futuras).



Diseñar para el cambio

- El software cambia.
- Para anticiparse a los cambios en los requisitos hay que diseñar pensando en qué aspectos pueden cambiar.
- Los patrones de diseño están orientados al cambio.

1. Patrones

Cómo llegar a ser un maestro de ajedrez

- Primero aprender las reglas del juego, nombres de las piezas, movimientos legales, geometría y orientación del tablero, etc.
- Luego aprender los principios relativos al valor de las piezas, valor estratégico de las casillas centrales, jaque cruzado, etc.

1. Patrones

- Pero… para llegar a ser un maestro, hay que estudiar los diseños de otros maestros.
- Estos diseños contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente.
- Hay cientos de estos patrones.

1. Patrones

Christopher Alexander (1977):

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y describe la esencia de la solución a ese problema, de tal modo que pueda utilizarse esta solución un millón de veces más, sin siquiera hacerlo de la misma manera dos veces.

1. Patrones

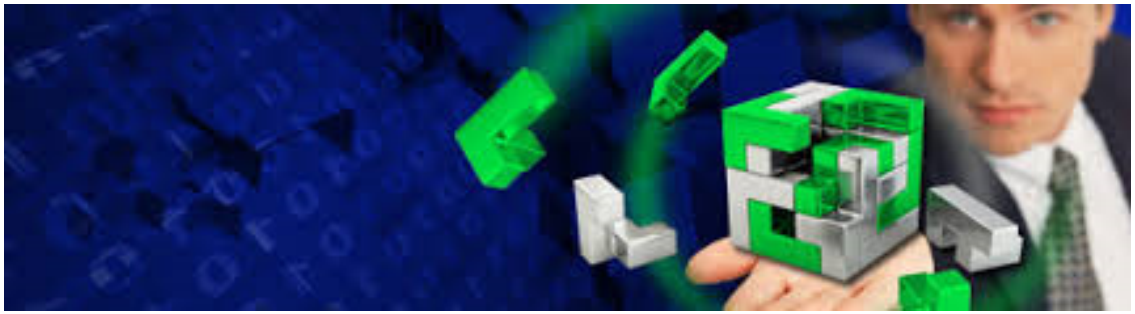
Un patrón:

- tiene un nombre.
- es una solución a un problema en un contexto particular recurrente.

Los patrones facilitan la reutilización de diseños y arquitecturas de software que han tenido éxito

2. Motivación de los Patrones

- Capturan la experiencia y la hacen accesible a los no expertos.
- Los nombres de patrones forman un vocabulario que ayuda a los desarrolladores a comunicarse mejor.
- Un sistema es comprendido más rápidamente cuando está documentado con los patrones que usa.



2. Motivación de los Patrones

- Los patrones pueden ser la base de un manual de ingeniería de software.

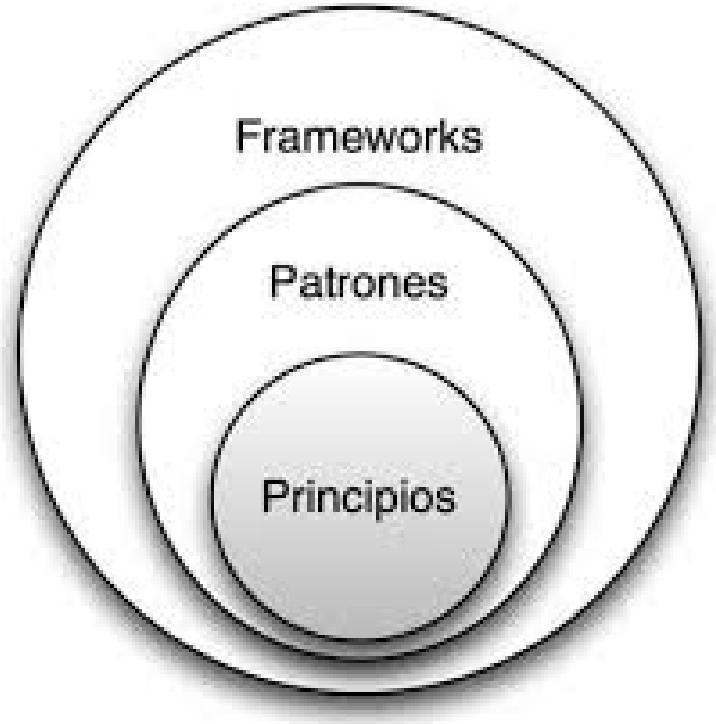
- Facilitan la reestructuración de un sistema tanto si fue o no concebido con patrones en mente.

2. Motivación de los Patrones

- El software cambia
  - Para anticiparse a los cambios, en los requisitos hay que diseñar pensando en qué aspectos pueden cambiar.
  - Los patrones de diseño están orientados al cambio.
- Reutilización
  - Los patrones de diseño soportan la reutilización de arquitecturas de software.
  - Los frameworks soportan la reutilización del diseño y del código.

3. Patrones y Frameworks

- Los patrones de diseño tienen descripciones más abstractas que los frameworks.



3. Patrones y Frameworks

- Las descripciones de patrones suelen ser independientes de los detalles de implementación o del lenguaje de programación (salvo ejemplos usados en su descripción).

- Los frameworks están implementados en un lenguaje de programación, y pueden ser ejecutados y reutilizados directamente.

3. Patrones y Frameworks

- Los patrones de diseño son elementos arquitecturales más pequeños que los frameworks.

  - Un framework incorpora varios patrones
  - Los patrones se pueden usar para documentar frameworks

- Los patrones de diseño están menos especializados que los frameworks.

  - Los frameworks siempre se aplican a un dominio de aplicación particular

4. Frameworks (Armazones)

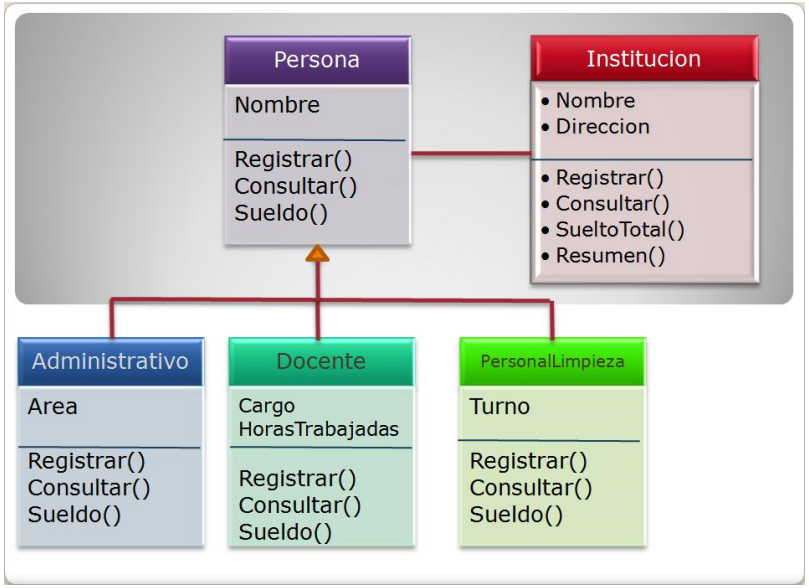
Características

- Un framework ofrece un conjunto integrado de funcionalidad específica de un dominio.

Ejemplo:

- aplicaciones financieras,
- servicios de telecomunicación,
- sistemas de ventanas,
- bases de datos,
- aplicaciones distribuidas,
- núcleos de SO

4. Frameworks

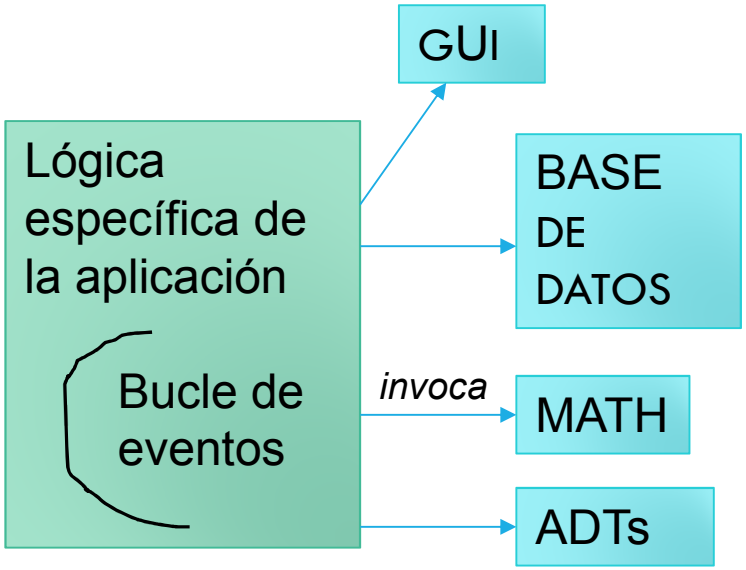


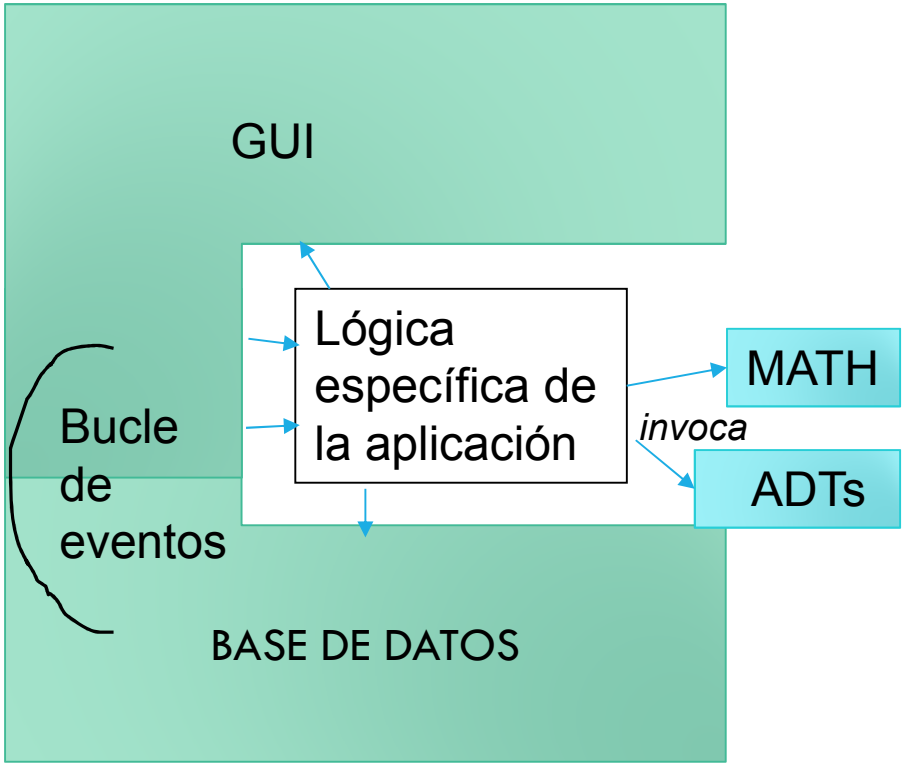
Características

- Un framework es una aplicación medio-acabada.
- Las aplicaciones completas se desarrollan mediante herencia, e instanciando componentes parametrizados del framework.

4. Frameworks

Diferencias con bibliotecas





Arquitectura basadas en bibliotecas de clases

Reutilización de código

Arquitectura basadas en framework

Reutilización de diseño y código

5. Patrones

Según el nivel de abstracción los patrones pueden clasificarse de la siguiente forma:

- **Patrones arquitecturales**
- **Patrones *de* diseño**
- **Patrones elementales (idioms)**

5. Patrones

- **Patrones arquitecturales**
  - Expresan un paradigma fundamental para estructurar un sistema de software.
  - Proporcionan un conjunto de subsistemas predefinidos, con reglas y guías para organizar las relaciones entre ellos.

Ejemplos

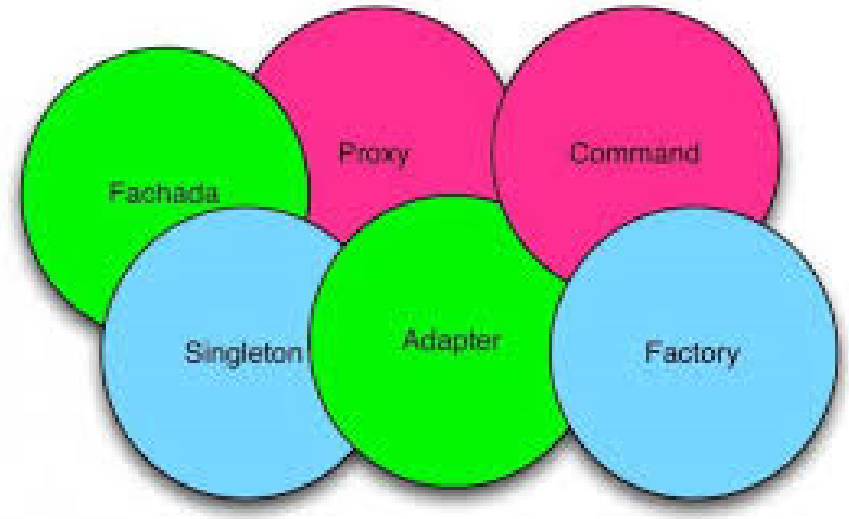
- Jerarquía de capas
- Tuberías y filtros
- Cliente/Servidor
- Maestro-Esclavo
- Control centralizado y distribuido

5. Patrones

- **Patrones de diseño**
  - Compuestos de varias unidades arquitecturales más pequeñas.
  - Describen el esquema básico para estructurar subsistemas y componentes.

Ejemplos

- Proxies
- Factorías
- Adaptadores
- Composición
- Broker



5. Patrones

- **Patrones elementales (idioms)**
  - Específicos de un lenguaje de programación.
  - Describen cómo implementar componentes particulares de un patrón.
  - Ejemplo: En Java implementar una interface en una clase anónima.

Ejemplos

- Modularidad
- Interfaces mínimas
- Encapsulación
- Objetos
- Acciones y Eventos
- Concurrencia

5.1 Patrones de diseño

	CREACION	ESTRUCTURALES	COMPORTAMIENTO
DE CLASE	Factory Method	Class Adapter	Template Method Interpreter
DE OBJETO	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsability Command Interpreter Iterator Mediator Memento Observer State Strategy Visitor

5.1 Patrones de diseño

- **Patrones de creación**
  Tratan la inicialización y configuración de clases y objetos.

- **Patrones estructurales**
  Tratan de desacoplar la interfaz e implementación de clases y objetos.

- **Patrones de comportamiento**
  Tratan las interacciones dinámicas entre sociedades de clases y objetos:
  ¿Cómo interaccionan y se distribuyen responsabilidades los objetos?

5.1.1 Patrones de creación

Los **patrones de creación** abstraen el proceso de instanciación de objetos, ayudando a que el sistema sea independiente de cómo se crean, componen y representan sus objetos

Estos patrones:

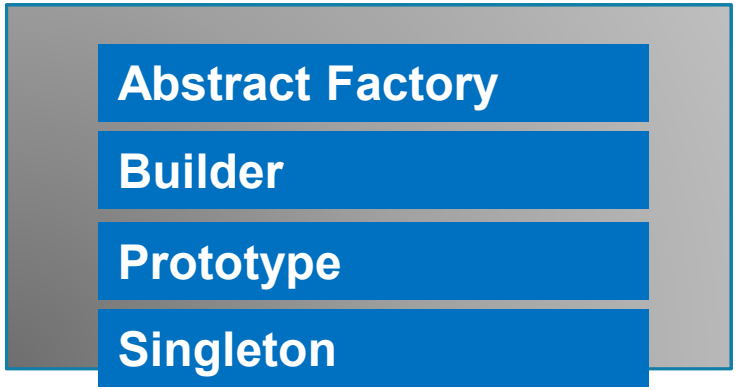
- Encapsulan el conocimiento sobre las clases concretas que utiliza el sistema.

- Brindan soporte a una de las tareas mas comunes dentro de la programación orientada a objetos: la instanciación.

5.1.1 Patrones de creación

- **De clase:** usa herencia para variar la clase del objeto creado.

- **De objeto:** delega la creación en otro objeto.

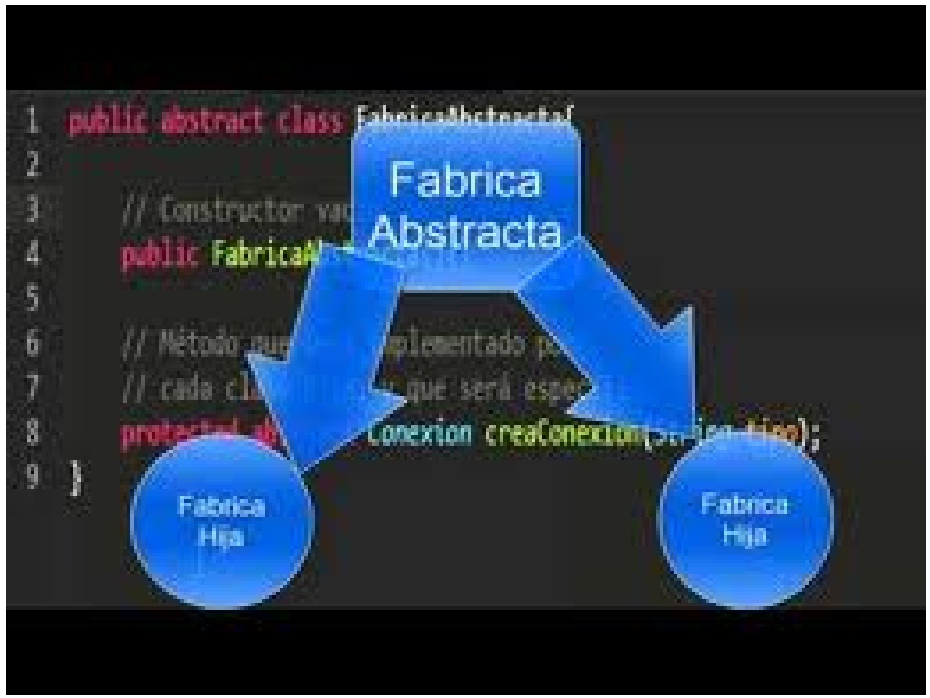


5.1.1 Patrones de creación

- **Factory Method**: define una interfaz para crear un objeto delegando la decisión de qué clase crear en las subclases. Este enfoque también puede ser llamado constructor “virtual”.

5.1.1 Patrones de creación

- **Factoría abstracta:** provee una interfaz para crear familias de objetos, productos relacionados o que dependen entre sí, sin especificar sus clases concretas.

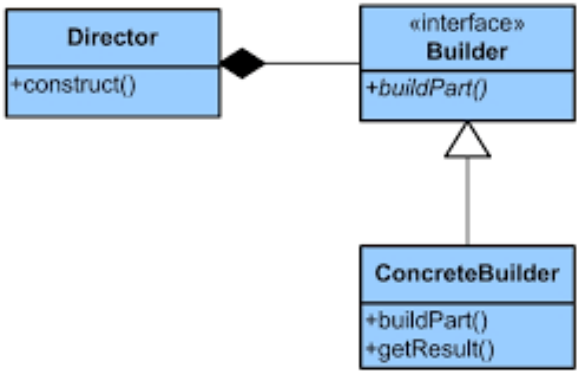


5.1.1 Patrones de creación

- **Builder:** Simplifica la construcción de objetos con estructura interna compleja y permite la construcción de objetos paso a paso.

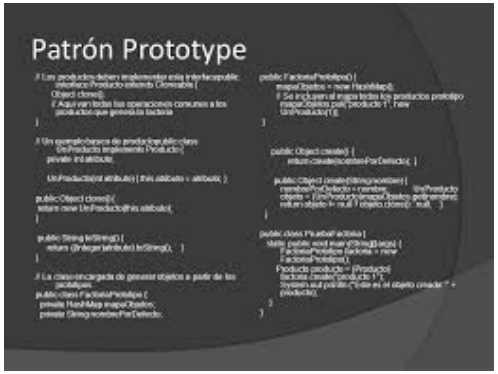
Ejemplo: este patrón se encuentra en los restaurantes de comidas rápidas que ofrecen menúes infantiles.

Los menúes están formados de: un plato principal, un acompañamiento, una bebida y un juguete. Si bien el contenido del menú puede variar, el proceso de construcción es siempre el mismo: el cajero indica a los empleados los pasos a seguir: preparar un plato principal, un acompañamiento, incluir un juguete y guardarlos en una bolsa. La bebida se sirve en un vaso y queda fuera de la bolsa.



5.1.1 Patrones de creación

- **Prototype:** facilita la creación dinámica de objetos mediante la definición de clases cuyos objetos pueden crear duplicados de si mismos. Estos objetos son llamados prototipos.



Ejemplo: un escenario frecuente es contar con GUIs (Interfaz Gráfica de Usuario) que cuenten con un gran número de controles similares, los cuales deben ser inicializados a un determinado estado común para mantener consistencia.

El proceso de inicialización se repite varias veces por cada control de manera que las líneas de código se incrementan. Con el fin de optimizar estas partes del código se puede contar con un objeto inicializado en un determinado estado estándar y luego obtener clones de él ya inicializados.

5.1.1 Patrones de creación

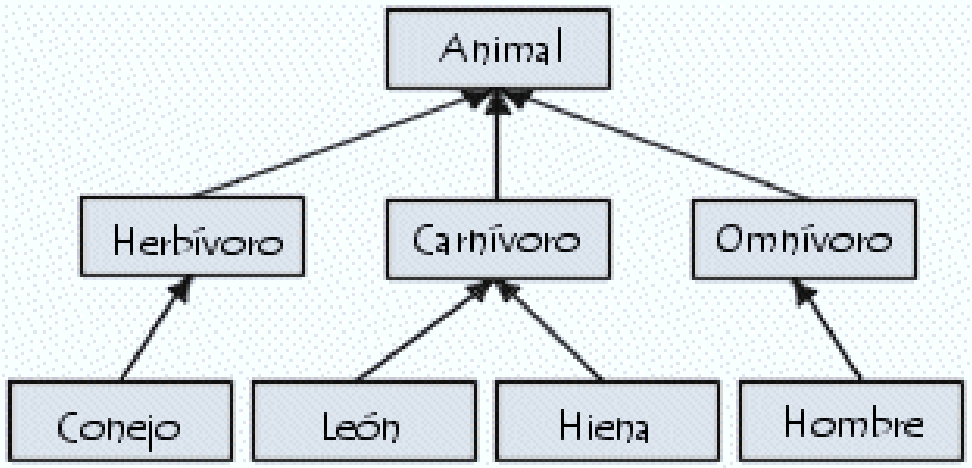
- **Singleton:** asegura que una determinada clase sea instanciada una y sólo una vez, proporcionando un único punto global a ella.
- **Object Pool:** Factoría que asegura que sólo hay un miembro (singleton) o un conjunto determinado (object pool) de una clase, y proporciona un punto global de acceso a él.

5.1.2 Patrones estructurales

- Se encargan de combinar clases y objetos parar formar estructuras más grandes.
- Los **patrones estructurales de clases** utilizan la herencia para componer interfaces o implementaciones.

Herencia múltiple

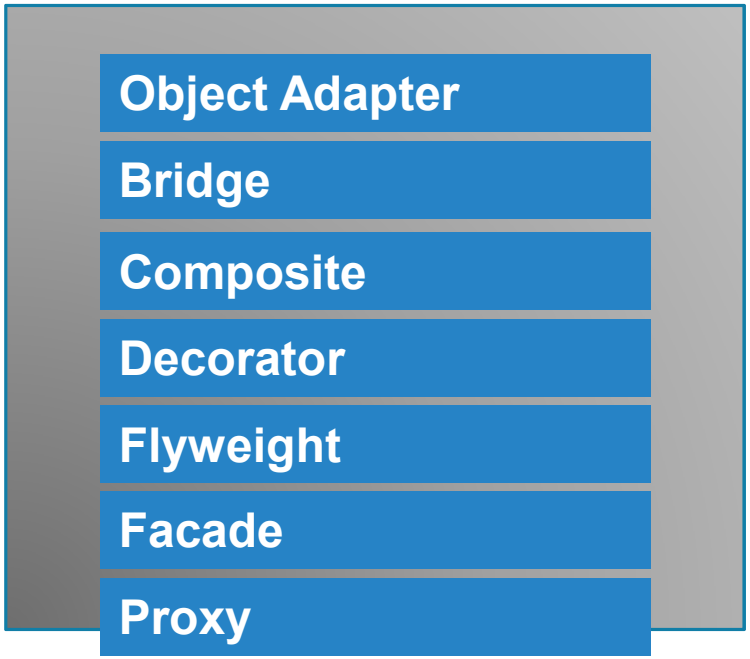
Class Adapter



5.1.2 Patrones estructurales

- Los **patrones estructurales de objetos** describen formas de componer objetos para obtener nuevas funcionalidades. La flexibilidad añadida mediante la composición de objetos está dada por la capacidad de cambiar la composición en tiempo de ejecución, que es imposible con la composición de clases.

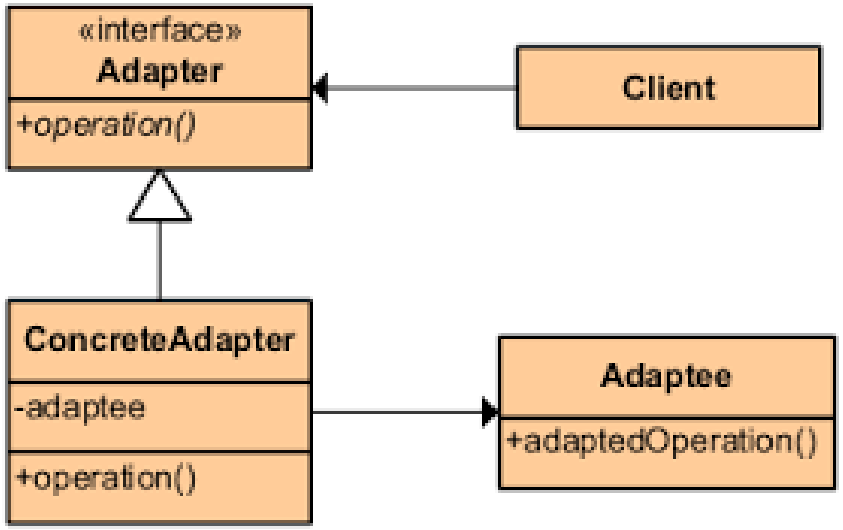
Ejemplos típicos son:
- cómo comunicar dos clases incompatibles
- cómo añadir funcionalidad a objetos.



5.1.2 Patrones estructurales

- **Adapter:** oficia de intermediario entre dos clases cuyas interfaces son incompatibles de manera tal que puedan ser utilizadas en conjunto.

Ejemplo: en la vida cotidiana se ven ejemplos de este patrón, el mas común de ellos es el de los adaptadores de enchufes los que permiten utilizar un enchufe de dos patas planas en un toma corriente de dos patas redondas.



5.1.2 Patrones estructurales

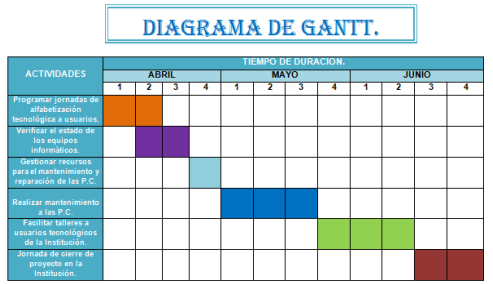
- **Bridge:** disocia un componente complejo en dos jerarquías de clases: una abstracción funcional y la implementación interna, para que ambas puedan variar independientemente.

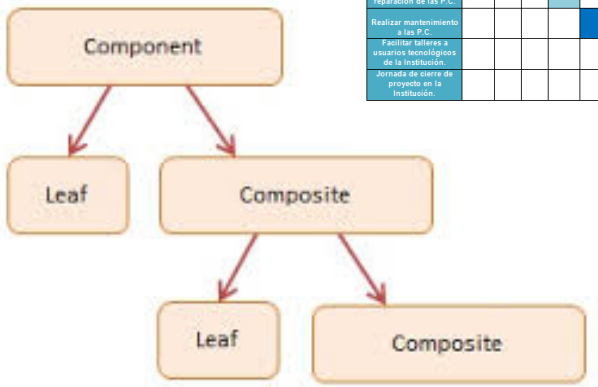
Ejemplo: los electrodomésticos y sus interruptores de encendido pueden ser considerados como ejemplos de este patrón donde el interruptor de encendido es considerado la abstracción y el electrodoméstico en si la implementación.

El interruptor podría ser un simple interruptor de encendido/apagado, un regulador de velocidades u alguna otra opción, mientras que el electrodoméstico puede ser una lámpara, un ventilador de techo, etc.

5.1.2 Patrones estructurales

- **Composite:** compone objetos en estructuras de árboles para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los complejos.



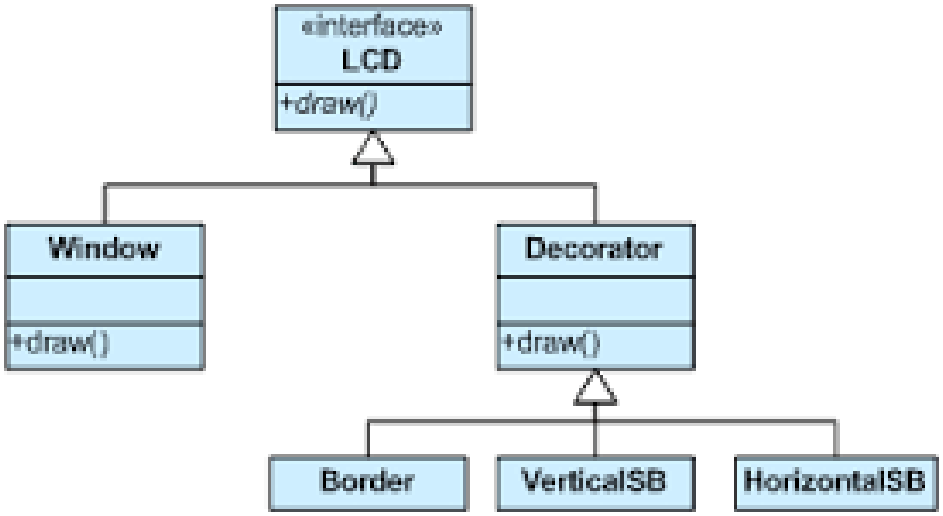


Ejemplo: en una gráfica de Gantt existen tareas simples (con una actividad) y compuestas (que contienen varias tareas). Modelar estos dos tipos de tareas en una jerarquía de clases donde ambas son subclases de una clase que cuente con un método “calculáTiempoUtilizado” permitiría tratar de forma uniforme a tareas simples y compuestas para calcular el tiempo utilizado por cada una de ellas. Concretamente una tarea simple informa el tiempo dedicado a ella, mientras que una compuesta lo hace sumando los tiempos insumidos de cada una de las tareas que contiene.

5.1.2 Patrones estructurales

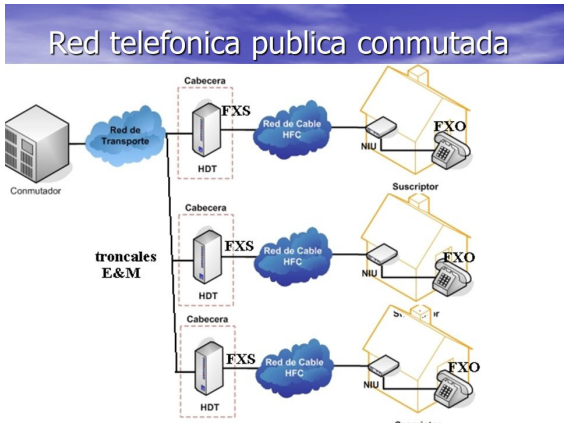
- **Decorator:** agrega o limita responsabilidades adicionales a un objeto de forma dinámica, proporcionando una alternativa flexible a la herencia para extender funcionalidad.

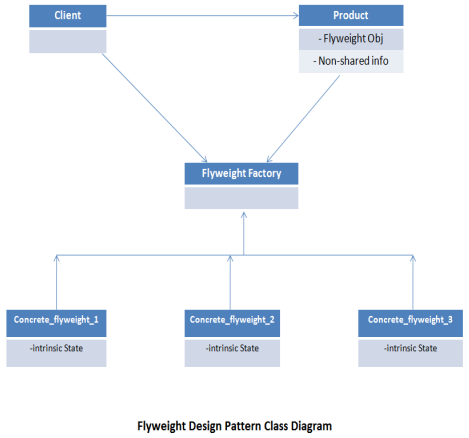
Ejemplo: si bien es cierto que se pueden colgar pinturas, cuadros y fotos en las paredes sin marcos, éstos suelen ser utilizados. Al momento de colgarse los cuadros junto con su marco pueden formar un solo "componente visual".



5.1.2 Patrones estructurales

- **Flyweight:** permite el uso de un gran número de objetos de grano fino de forma eficiente mediante compartimiento.



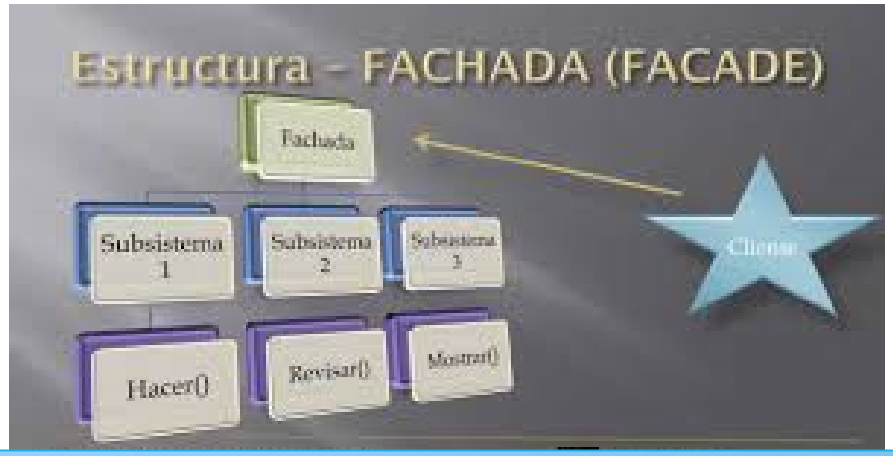


Flyweight Design Pattern Class Diagram

Ejemplo: La red telefónica pública conmutada es un ejemplo de este patrón ya que hay diversos componentes, por ejemplo nodos de conmutación, que se deben compartir entre los distintos usuarios. Los usuarios no conocen cuántos componentes de cada tipo hay disponibles al momento de realizar la llamada. Lo único por lo que se preocupan los usuarios es por obtener tono para marcar y efectuar la llamada.

5.1.2 Patrones estructurales

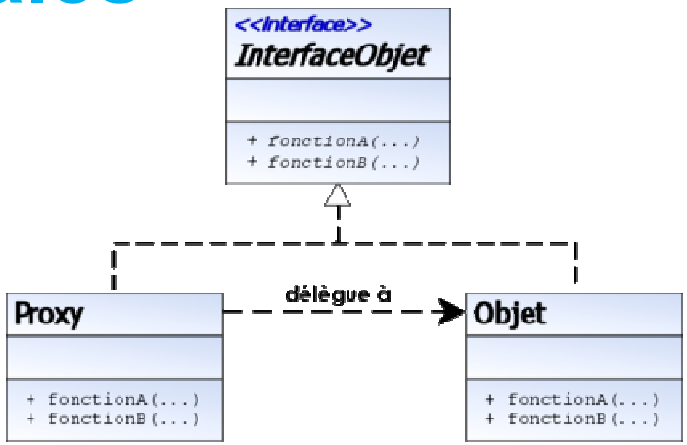
- **Facade:** proporciona una interfaz simplificada para un conjunto de interfaces de subsistemas. Define una interfaz de alto nivel que hace que un subsistema sea más fácil de usar.

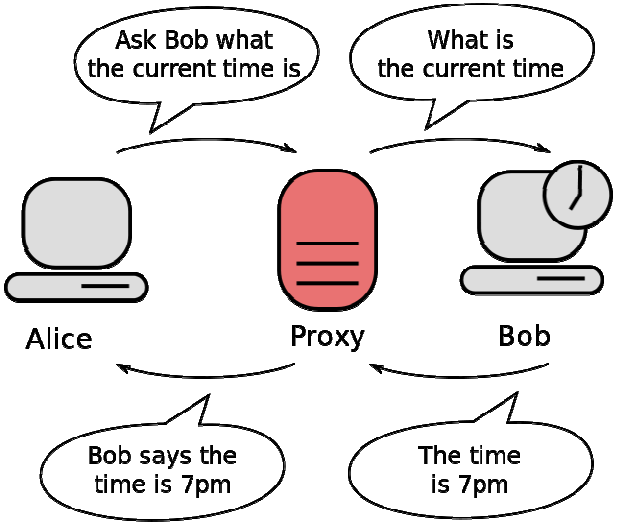


Ejemplo: En un sistema de compras los clientes contactan a un responsable de ventas que actúa como Facade al momento de realizar un pedido. Este representante de ventas actúa como Facade proveyendo una interface con los departamentos (subsistemas) de pedidos, facturación y envíos.

5.1.2 Patrones estructurales

- **Proxy:** Provee un sustituto o representante de un objeto para controlar el acceso a éste. Este patrón posee las siguientes variantes:





- **Proxy remoto:** se encarga de representar un objeto remoto como si estuviese localmente.
- **Proxy virtual:** se encarga de crear objetos de gran tamaño bajo demanda.
- **Proxy de protección:** se encarga de controlar el acceso al objeto representado.

5.1.3 Patrones de comportamiento

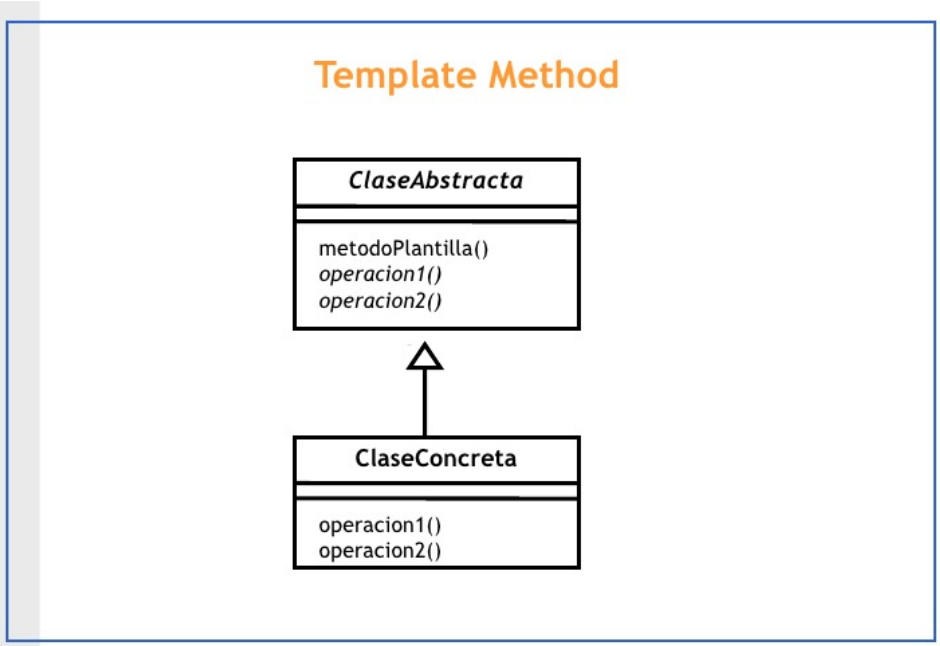
- Los patrones de comportamiento tienen que ver con:
  - definición de abstracciones de algoritmos
  - asignación de responsabilidades entre objetos
  - patrones de comunicación entre objetos
  - colaboraciones entre objetos para realizar tareas complejas reduciendo las dependencias
- Los patrones de comportamiento basados en **clases** utilizan la **herencia** para distribuir el comportamiento entre clases, ellos son:

Template Method

Interpreter

5.1.3 Patrones de comportamiento

- **Template method:** Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.



5.1.3 Patrones de comportamiento

- **Interpreter:** Cuando una clase de problemas se repite en un dominio bien conocido, se pueden caracterizar estos problemas como un lenguaje y pueden ser tratados por un "motor" de interpretación. Este patrón busca definir un intérprete para dicho lenguaje, para el cual define una gramática y un intérprete de la misma para poder resolver los problemas.

Ejemplo: distintos motores de bases de datos (Oracle, SQL Server, Sybase, DB2, etc.) utilizan distintos códigos de error para indicar fallas (errores de clave duplicada, longitud de datos, etc.). La utilización de éste patrón permitiría definir un intérprete de errores para cada motor de base de datos con el cual se determinaría la falla y tomarían las acciones pertinentes en función de la misma.

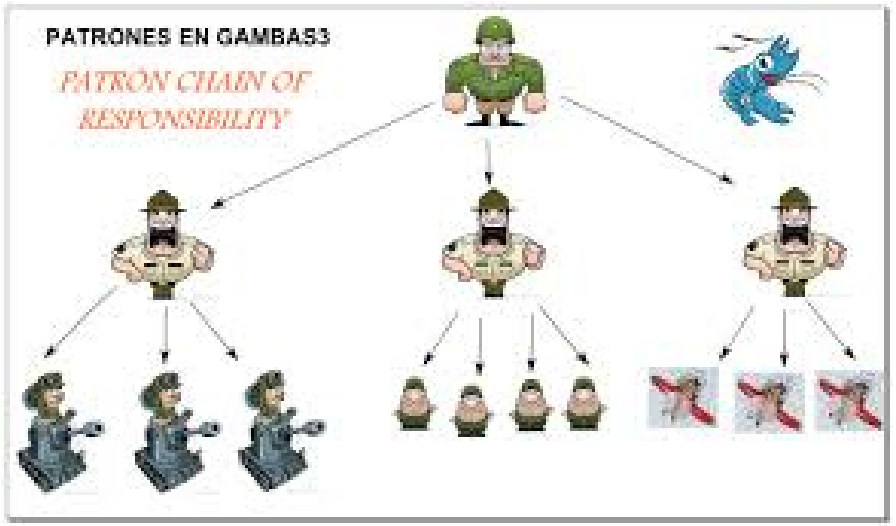
5.1.3 Patrones de comportamiento

- Los patrones de comportamiento basados en **objetos** utilizan la **composición**.

5.1.3 Patrones de comportamiento

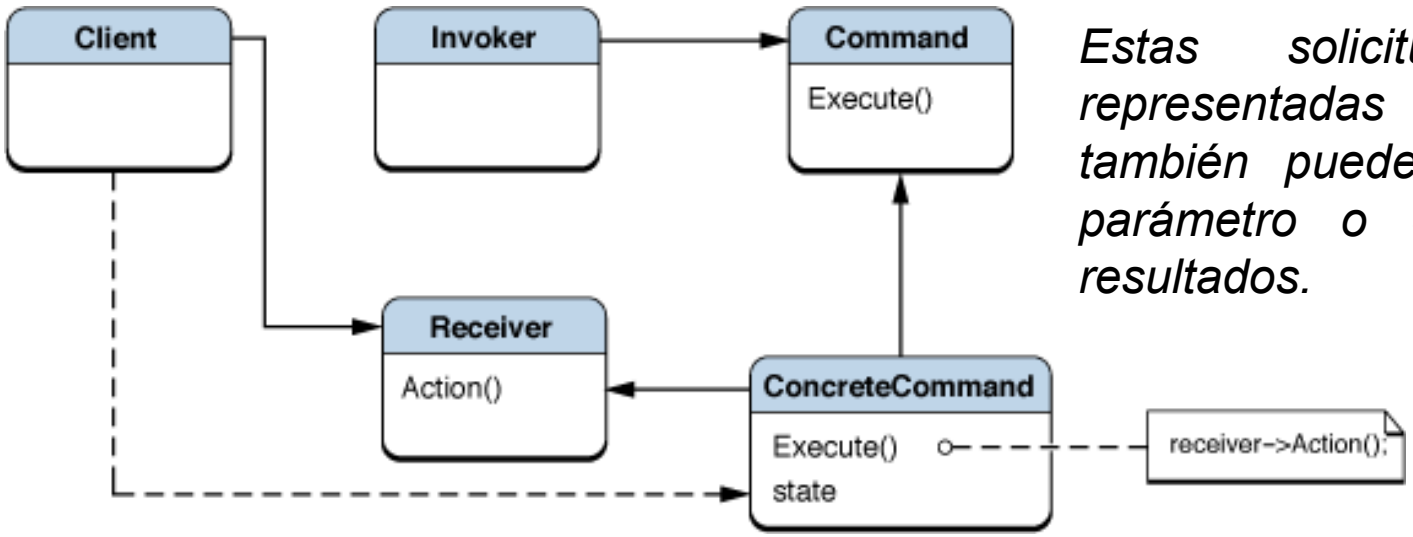
- **Chain of responsability:** Petición delegada al proveedor de servicio responsable. Establece una cadena de mensajes dentro del sistema, de manera tal que dicho mensaje sea manejado en el mismo nivel donde fue emitido, o redirigido a un objeto capaz de manejarlo.

Evita acoplar el emisor del mensaje con un receptor, dando a más de un objeto la posibilidad de responder al mensaje.



5.1.3 Patrones de comportamiento

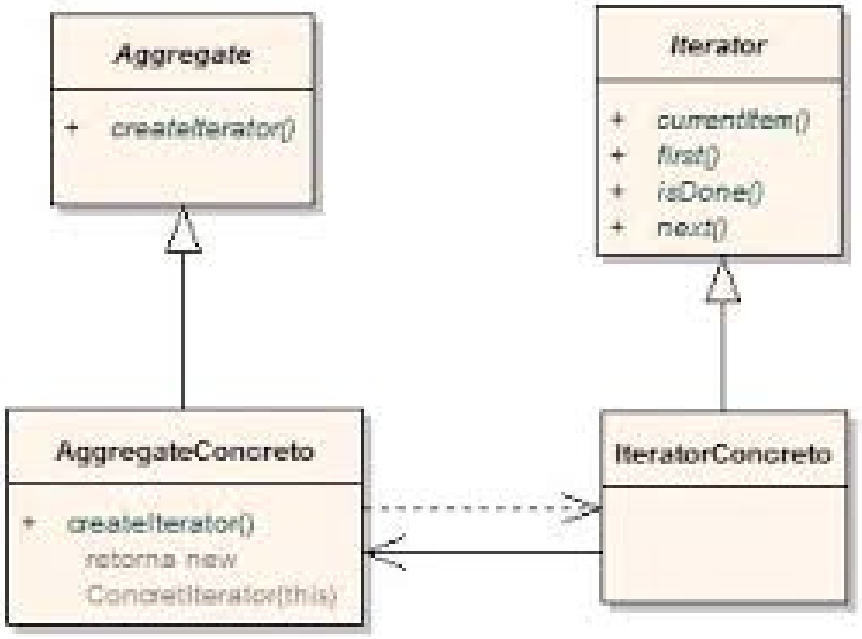
- **Command:** Encapsula una petición como un objeto. Representa una solicitud con un objeto, de manera tal de poder parametrizar a los clientes con distintas solicitudes, encolarlas o llevar un registro de las mismas, y poder deshacer las operaciones.



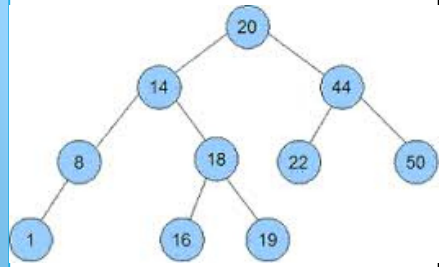
Estas solicitudes, al ser representadas como un objeto también pueden pasarse como parámetro o devolverse como resultados.

5.1.3 Patrones de comportamiento

- **Iterator:** provee un modo de acceder secuencialmente a los elementos de un objeto agregado (una colección) sin exponer su representación interna. El iterador está altamente acoplado al objeto agregado.

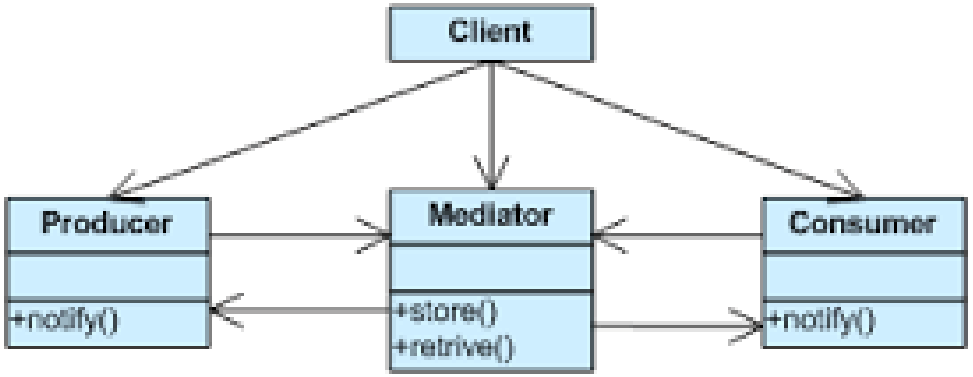


Ejemplo: Los árboles-B pueden recorrerse de tres formas distintas: pre-orden, en-orden y post-orden. La aplicación de este patrón permitiría definir un iterador para cada tipo de recorrido, pudiendo ser utilizados para recorrer el árbol sin exponer su contenido.



5.1.3 Patrones de comportamiento

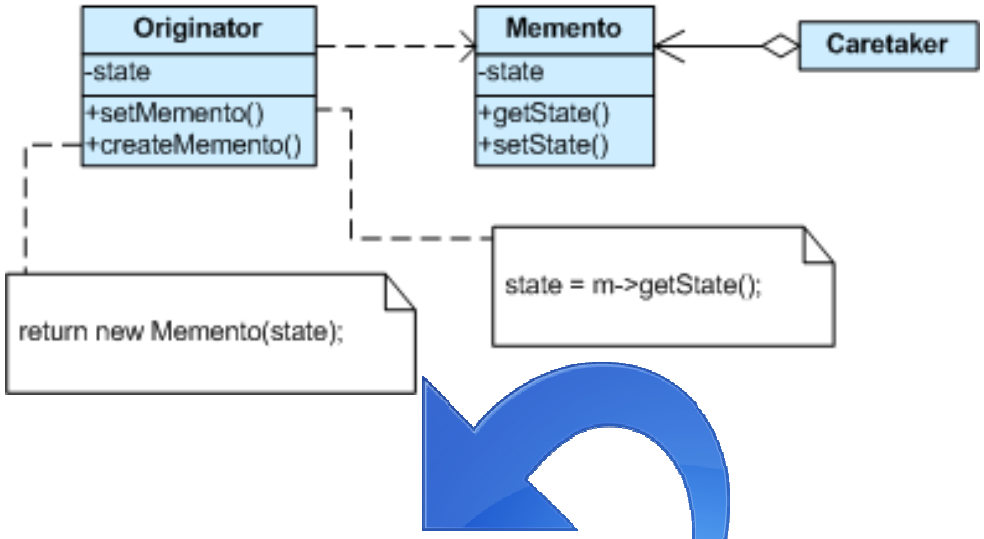
- **Mediator:** simplifica la comunicación entre objetos dentro del sistema mediante la introducción de un objeto mediador que administra la distribución de mensajes entre objetos.



Ejemplo: Este patrón puede verse en las torres de control de los aeropuertos. Los pilotos de los aviones que se encuentran por despegar o aterrizar se comunican con la torre en lugar de hacerlo explícitamente entre ellos. La torre de control regula quien puede aterrizar y despegar, pero no se encarga de controlar todo el vuelo.

5.1.3 Patrones de comportamiento

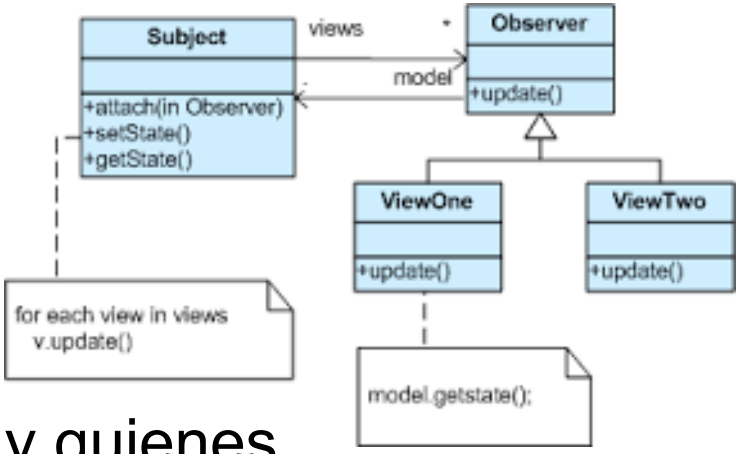
- **Memento:** preserva una "fotografía instantánea" del estado de un objeto con el fin de permitirle volver a su estado original, sin revelar su contenido al mundo exterior.



Ejemplo: permite implementar la funcionalidad "Deshacer" o "Undo", funcionalidad muy importante de los editores de texto. Para ello ante cada cambio del documento se debe tomar una nueva fotografía del estado del documento antes de la modificación, para poder volver al mismo. Deben tenerse en cuenta: el orden de guardado de los cambios, y tiempo de limpiar el registro de estados intermedios porque puede consumir muchos recursos.

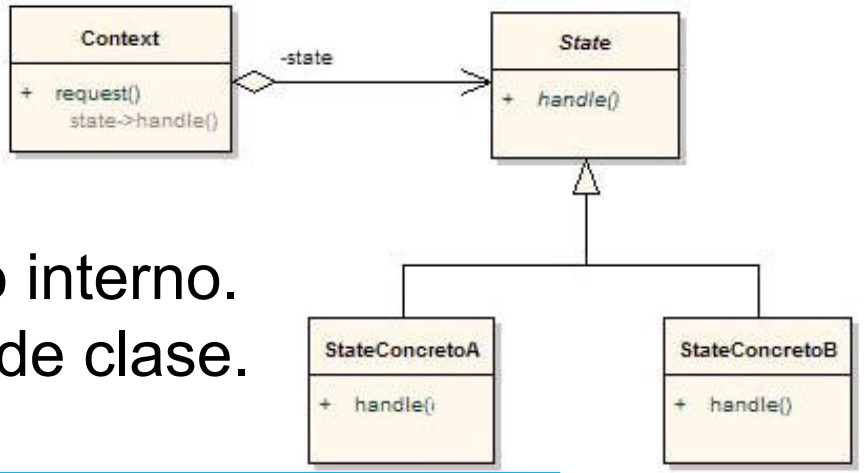
5.1.3 Patrones de comportamiento

- **Observer:** permite a un componente transmitir de forma flexible mensajes a los objetos que hayan expresado interés en él. Estos mensajes se disparan cuando el objeto ha sido actualizado, y quienes hayan expresado interés reaccionen ante este evento.



Ejemplo: es usado en las subastas, cada ofertante (Observer) tiene un indicador con su número, este es utilizado para indicar la aceptación de una oferta. El subastador (Subject, objeto observado) comienza la subasta con una oferta inicial, cuando un ofertante toma esa oferta el subastador les transmite a todos los ofertantes que el precio ha cambiado.

5.1.3 Patrones de comportamiento

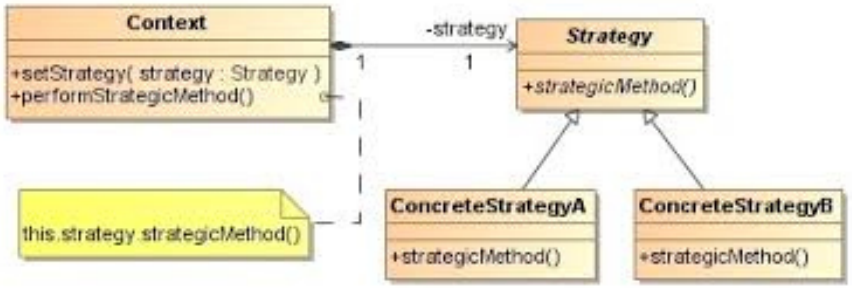


- **State:** permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. El objeto parecerá que cambió de clase.

Ejemplo: este patrón se observa en las máquinas expendedoras de golosinas, las cuales pasan por distintos estados: stock disponible, dinero depositado, capacidad para dar vuelto, golosina seleccionada, etc. En cada uno de éstos estados la máquina se comporta diferente. Cuando se deposita dinero y se elije una golosina, la expendedora puede entregar un producto y no cambiar su estado, entregar un producto y cambiar su estado (por ejemplo quedarse sin stock).

5.1.3 Patrones de comportamiento

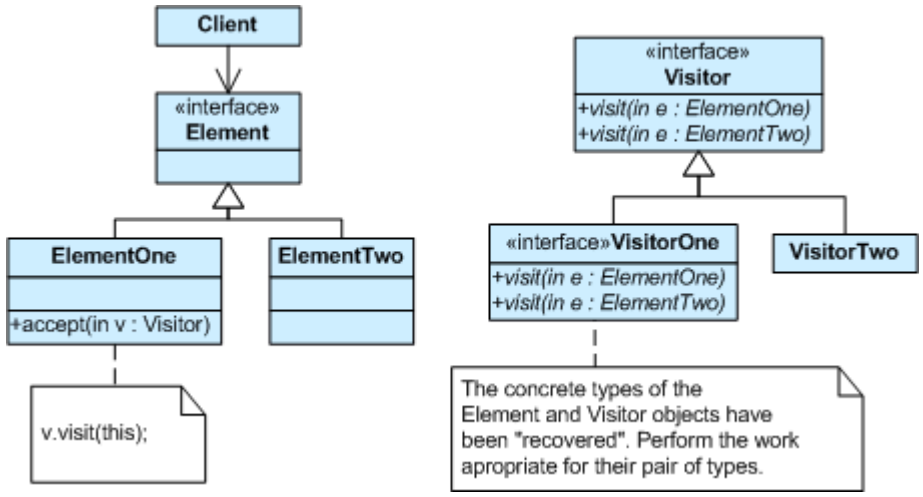
- **Strategy:** define una jerarquía de clases que representan algoritmos, los cuales pueden ser intercambiados por la aplicación en tiempo de ejecución.



Ejemplo: Se dispone de un programa que encripta y desencripta mensajes de texto usando distintos algoritmos de encriptación. Cada uno de estos algoritmos de encriptación puede modelarse como una clase con servicios de encriptación (un método "encriptáMensaje" que recibe el texto llano y una clave, para devolver un texto cifrado y servicios de desencriptación (un método "desencriptáMensaje" que recibe una clave y un texto cifrado para devolver uno descifrado).

5.1.3 Patrones de comportamiento

- **Visitor:**
  - Representa una operación sobre elementos de una estructura de objetos.
  - Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.
  - Brinda una forma sencilla y mantenible de realizar acciones sobre una familia de clases.



5.1.3 Patrones de comportamiento

- **Visitor:**

Ejemplo: un compilador interpreta código fuente y lo representa como un árbol de sintaxis abstracta (Abstract Syntax Tree, AST), el cual cuenta con diversos tipos de nodos (asignaciones, expresiones condicionales, etc.).
Sobre este árbol se desean ejecutar algunas operaciones como: revisar que todas las variables fueron declaradas, chequeos de tipos de datos, generación de código, etc.
Una forma de realizar estas operaciones es mediante la implementación del patrón Visitor, el cual recorrerá toda la estructura del árbol. Cuando un nodo acepte al Visitor, éste invocará al método de visita definido en el Visitor que toma por parámetro al nodo siendo visitado.