

Programación 2

- Es una materia de Teoría y práctica
- Modalidad: 1 parcial largo (mitad de octubre) , multiple choice practico y teorico
- Trabajo práctico integrador: en grupo, armar un diagrama
- Se promociona con 8 para arriba en la nota de cursada, nos anotamos en el final pero solo nos presentamos SINO con final obligatorio
- Material de apoyo: en el aula *anotar cositas porque hay estan las respuestas a los parciales*
- Tipos de diagramas que usaremos
 - Diagrama de clases
 - Diagrama de secuencia

luis.alvarez@ort.edu.ar

<u>Listado de patrones:</u> State` Strategy` Singleton` Composite` Adapter` Observer` Factory Method` Proxy` Decorator` Memento`	-Si nos da tiempo vemos estos: -Abstract Factory: Busca agrupar un conjunto de clases que tiene un funcionamiento en común llamadas familias, las cuales son creadas mediante un Factory. -Prototype -Builder
--	--

Consejos:

*Siempre se arranca por las funcionalidades y después por las características

*Entender siempre para qué lo estoy haciendo

Principio SOLID

Son una serie de normas o recomendación que guían la forma de programar, tiene 5 principios

- Código más mantenible
- Fácil de cambiar
- Muy extensible
- Responsabilidad única
 - Indica que cada clase debe tener una sola responsabilidad, debería encargarse de una sola parte del sistema, el objetivo es conseguir que las clases hagan una sola cosa
- Abierto cerrado
 - una entidad debe quedarse abierta para su extensión pero cerrada para su modificación
 - lo que significa es que para añadir funcionalidades se escribe nuevo código no se modifica código viejo

- se usa la herencia y el polimorfismo
 - Sustitución de Liskov
 - Toda clase que es hija de otra clase debe poder usarse como si fuera el mismo padre
 - Segregación de la interfaz
 - es mejor tener muchas clases pequeñas y especializadas
 - Inversión de las dependencias
 - los módulos de alto nivel no deben depender de los de bajo nivel, ambos deberían depender de interfaces
 - las implementaciones concretas no deberían depender de otras implementaciones concretas sino de capas abstractas, esto nos permite reducir el desacople entre sistemas de software A
-

Buenas prácticas

- **Cohesión y acoplamiento**
 - Alta Cohesión y un Bajo acoplamiento (alta relación entre las tareas y una baja dependencia).
- **SOLID**
 - Single Responsibility: Una clase debe tener una única razón para cambiar.
 - Open Closed: deben estar abiertas para ser extendidas y cerradas para ser modificadas.
 - Liskov Substitution: Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.
 - Interface Segregation: Muchas interfaces cliente específicas son mejores que una interfaz de propósito general.
 - Dependency inversion principle: Se debe depender de abstracciones (es decir, qué quiero representar) y no depender de implementaciones (es decir, cómo lo voy a representar). Ejemplo: Regulador, Termómetro, Horno, Alarma, Sensor, Bocina.

Acoplamiento

A veces hay acoplamiento entre métodos de la misma clase. Si tengo que conocer algo ya hay acoplamiento

Cohesión

Relacionado con la abstracción

UML

Tipo de relaciones:

- Generalización / Especialización
 - Herencia
 - Interfaz
- Asociación: No es una relación fuerte. Los tiempos de vida de los objetos son independientes.
 - Unidireccional: Propietario, Inmueble.
 - Bidireccional: Esposa, Esposo.
- Agregación: Variante de asociación... ... lo tiene tiempo de vida fuerte.
 - Carpeta, documento.
- Composición: Es un tipo de agregación... ... Cada componente de una agregación puede pertenecer "tan sólo a un todo". Tiempo de vida de la clase contenida, depende de la contenadora.
 - Persona, Órgano.
- Uso: Una clase utiliza parámetro para una de sus operaciones. El tiempo de vida de la clase, está ligada a la operación.
 - MP3, Reproductor MP3.

Patrones de Diseño

Cada patrón tiene:

1. Nombre
 - a. Describe en una o dos palabras un problema de diseño junto con sus soluciones y consecuencias.
 - b. Ejemplos: Observer (Observador), Strategy (Estrategia), Abstract Factory (Fábrica Abstracta)
2. Problema
 - a. Describe cuándo aplicar el Patrón.
 - b. Explica el problema y su contexto.
 - c. A veces el problema incluye una serie de condiciones que deben darse para que tenga sentido aplicar el Patrón.
3. Solución
 - a. Describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones.
 - b. La solución no describe un diseño en concreto, sino que un patrón es como una plantilla que puede aplicarse en muchas situaciones diferentes.
4. Consecuencias de implementar el patrón
 - a. Son los resultados, las ventajas e inconvenientes de aplicar el Patrón.
 - b. Son fundamentales para evaluar las alternativas de diseño y comprender los costos y beneficios de aplicar el Patrón.
 - c. Las consecuencias de un patrón incluyen su impacto sobre la flexibilidad, extensibilidad y portabilidad de un sistema.

Clasificación de los patrones:

- **Criterios** → 1) Propósito → Que hace un patrón / 2) Ámbito → especifica si se aplica a clases (estático) u objetos (dinámico)
- **Propósito**
 - De creación → tiene que ver con el proceso de creación de los objetos
 - Estructural → Tratan con la composición de clases y objetos
 - Comportamiento → Caracterizan el modo en que las clases y objetos interactúan y se reparten las responsabilidad
- **Ámbito**
 - Clase → se ocupan de las relaciones entre las clases y sus subclases, se establecen a través de la herencia (relaciones estáticas) fijadas en tiempo de compilación
 - Objeto → tratan las relaciones entre objetos que son DINÁMICAS ya que pueden compilar en tiempo de ejecución

Tienen diferentes formas de implementarlos

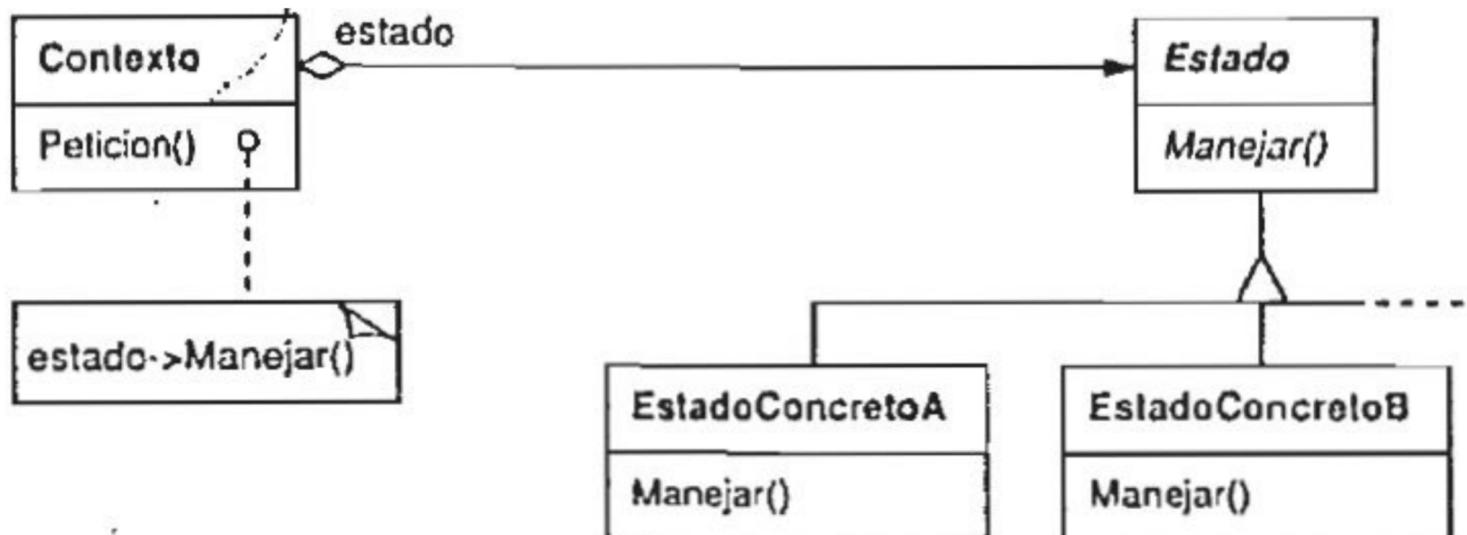
los patrones son una forma común de hacer algo / una plantilla que puede aplicarse en situaciones diferentes

Patrón State

Modifica el comportamiento cada vez que cambia el estado interno. Ejemplo: ConexionTCP. Establecida, escuchando y cerrando.

Aplicabilidad:

- El comportamiento depende del estado del objeto.
- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto



Implementación

DIAGRAMA DE USO

Patrón Strategy

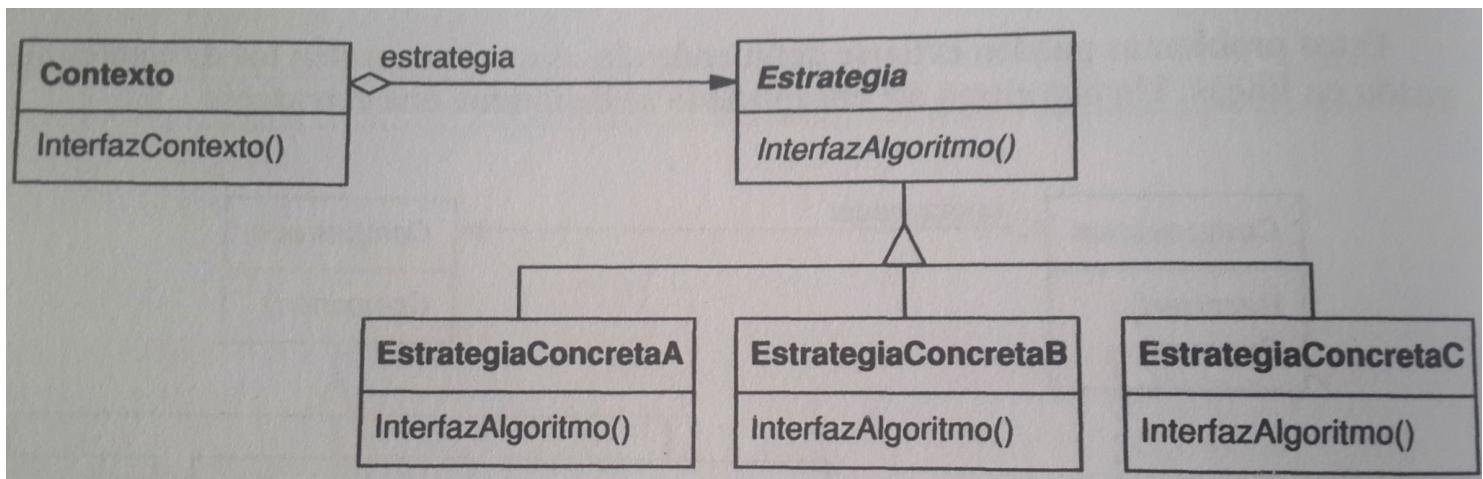
Propósito: Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables.

Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Motivación

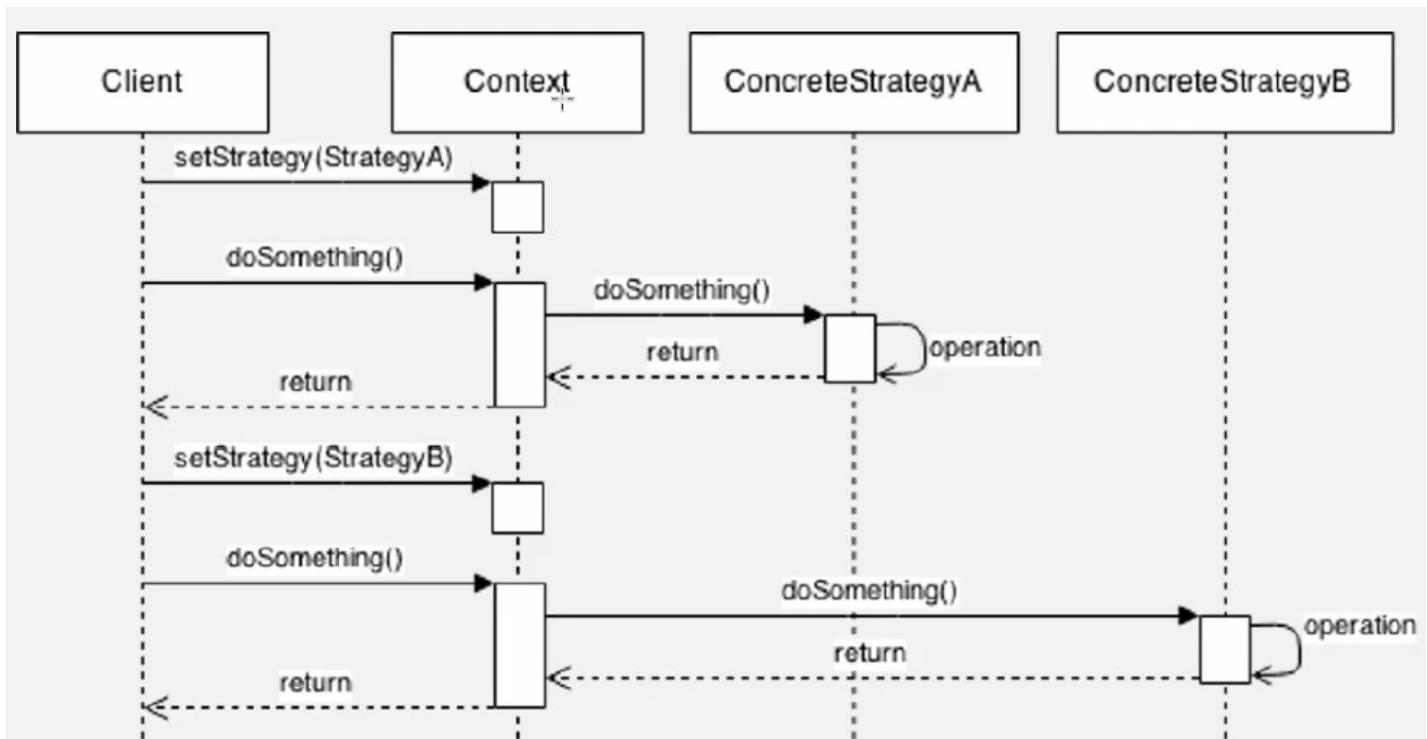
- El cliente sí conoce la implementación de un algoritmo complejo se complejiza
- Algoritmos son útiles en determinados momentos
- Difícil cambiar algoritmo si está acoplado a un cliente

Estructura



Estrategia: Declara una interfaz común a todos los algoritmos permitidos. El contexto usa esta interfaz para llamar al algoritmo definido por una Estrategia Concreta.

Secuencia



tiene variadas formas de implementarse

Patrón Singleton

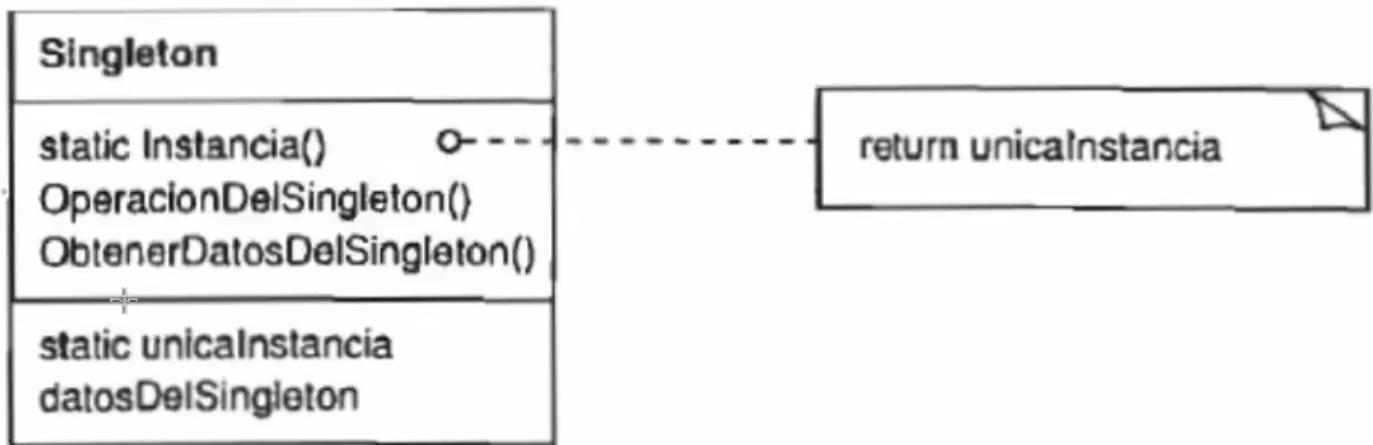
Propósito → Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella. se suele usar para base de datos, etc

Motivación

- Es importante que algunas clases tengan exactamente una instancia.
- La propia clase será responsable de su única instancia.

- La clase puede garantizar que no se puede crear ninguna otra instancia, y puede proporcionar un modo de acceder a la instancia.

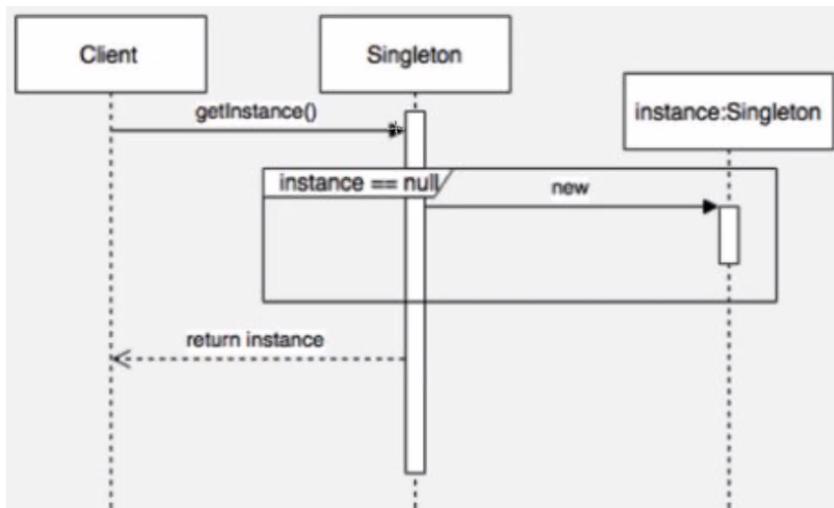
Estructura



Cuándo aplicarlo...>

- Cuando debe haber exactamente una instancia de una clase, y esta debe ser accesible a los clientes desde un punto de acceso conocido.

Secuencia



Implementación

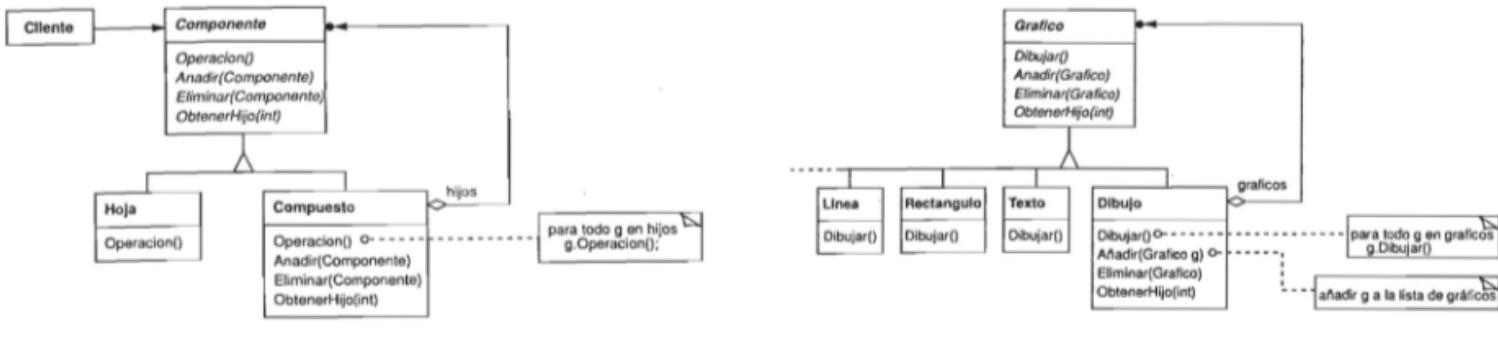
Patrón Composite

Propósito → representación de la estructura de objetos simples y complejos (contenedores o grupos de objetos) en estructuras de árbol para representar jerarquías de parte-todo en la que un objeto es siempre, un parte de un todo, o un todo compuesto por varias partes.

Motivación

- Debe existir una jerarquía clara bien definida.
- Identificar objetos simples y complejos o contenedores (Grupo de objetos simples).

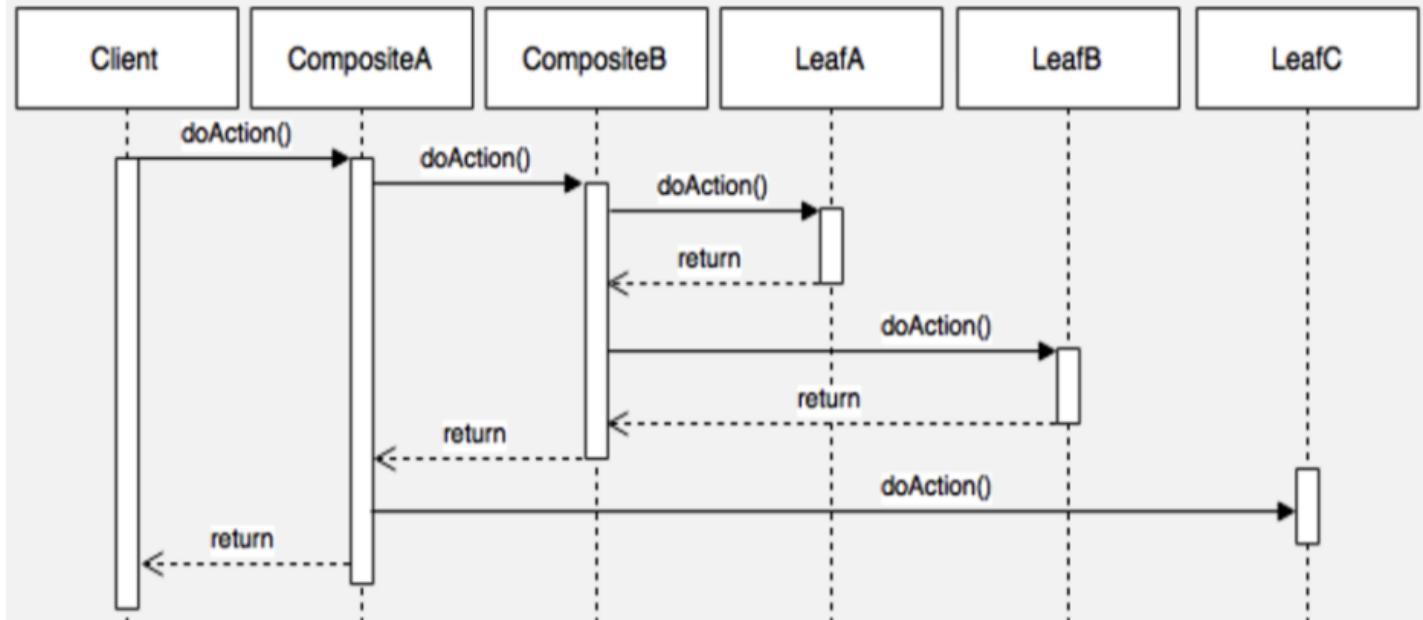
- No nos debe importar la forma de implementación de los arreglos de objetos completos



Estructura

Implementación

Composite pattern – Diagram of sequence

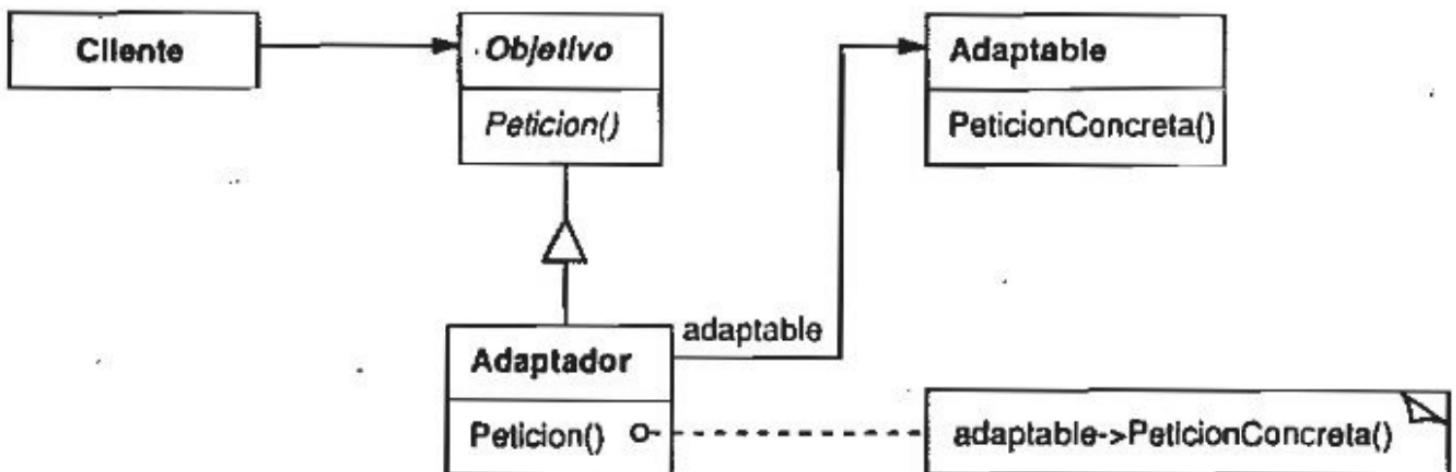


Patrón Adapter

Propósito: Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.

Motivación: Utilizar clases existentes en una aplicación que espera otro tipo de interfaces. Esto se logra adaptando la clase necesaria mediante una del dominio, que se encargará de adaptar el funcionamiento.

- Puede tener varios niveles de implementación
- Puede ser solo un pasamanos, redirigiendo la petición a los métodos de Adapter
- puede agregar funcionalidad para que la conversión sea posible (ejemplo: unificar dos string para un método que solo recibe un único string)
- puede agregar métodos extras además de los que debe convertir

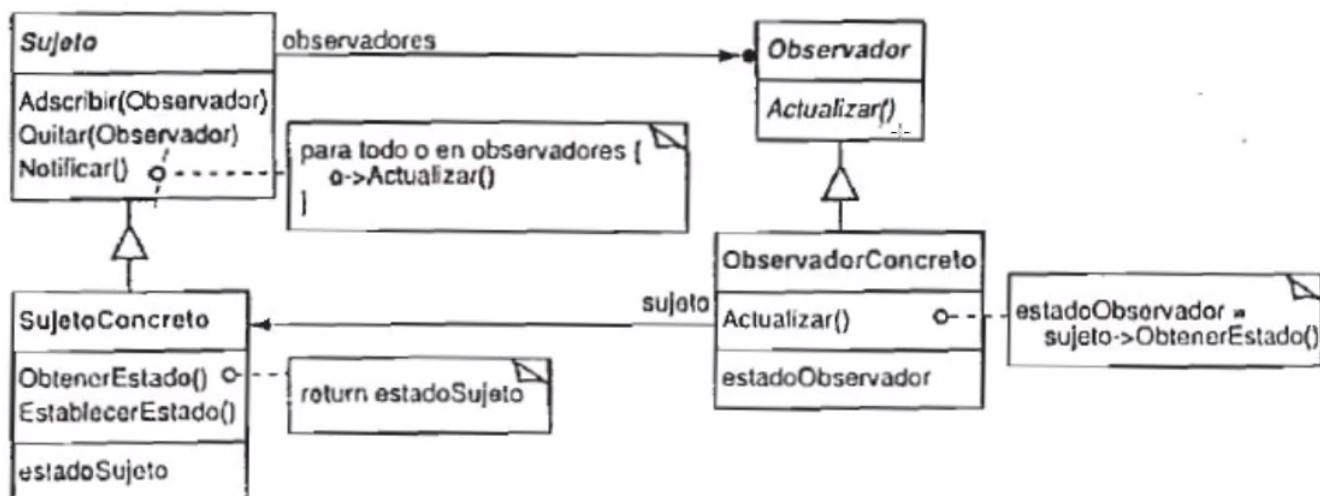


Patrón Observer

-Propósito: Define una dependencia de uno a muchos entre objetos, de forma que cuando el objeto cambie

de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

-Motivación: Reutilización independiente de clases que definen los datos y las representaciones



Patrón Factory method

Propósito: Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

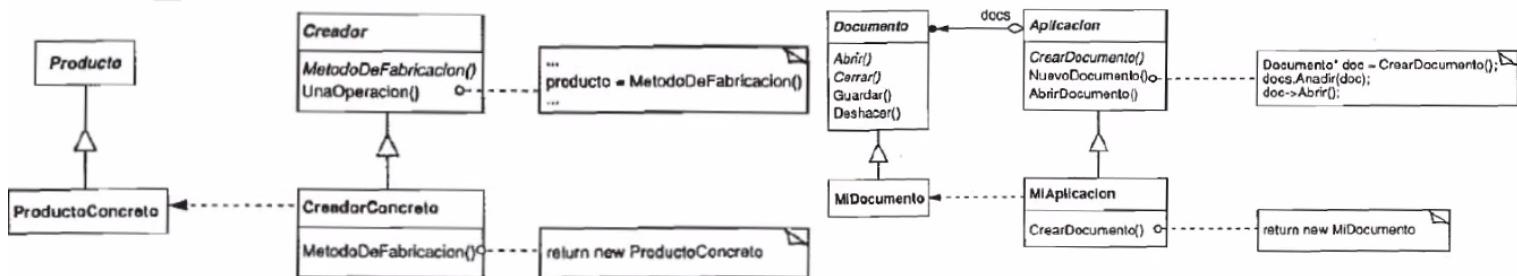
-Permite la creación de objetos de un subtipo determinado a través de una clase Factory.

Motivación → Permite la creación de objetos de un subtipo determinado a través de una clase Factory.

Cuando usar

- Una clase no puede prever la clase de objetos que debe crear.
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.

- Las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar concretamente en qué subclase de auxiliar se delega.



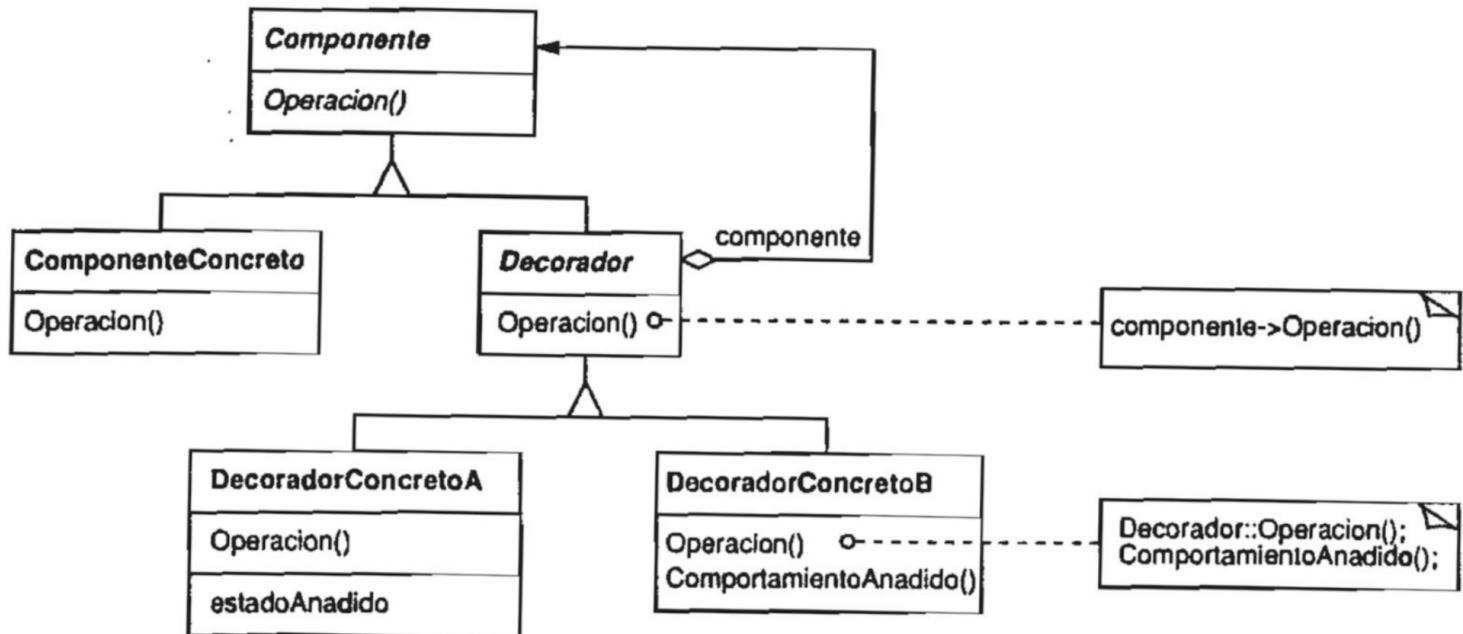
- Producto (Documento)**
 - Define la interfaz de los objetos que crea el método de fabricación.
- ProductoConcreto (MiDocumento)**
 - Implementa la interfaz Producto.
- Creador (Aplicacion)**
 - Declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConcreto.
 - Puede llamar al método de fabricación para crear un objeto Producto.
- CreadorConcreto**
 - Redefine el método de fabricación para devolver una instancia de un ProductoConcreto.

Patrón Decorator

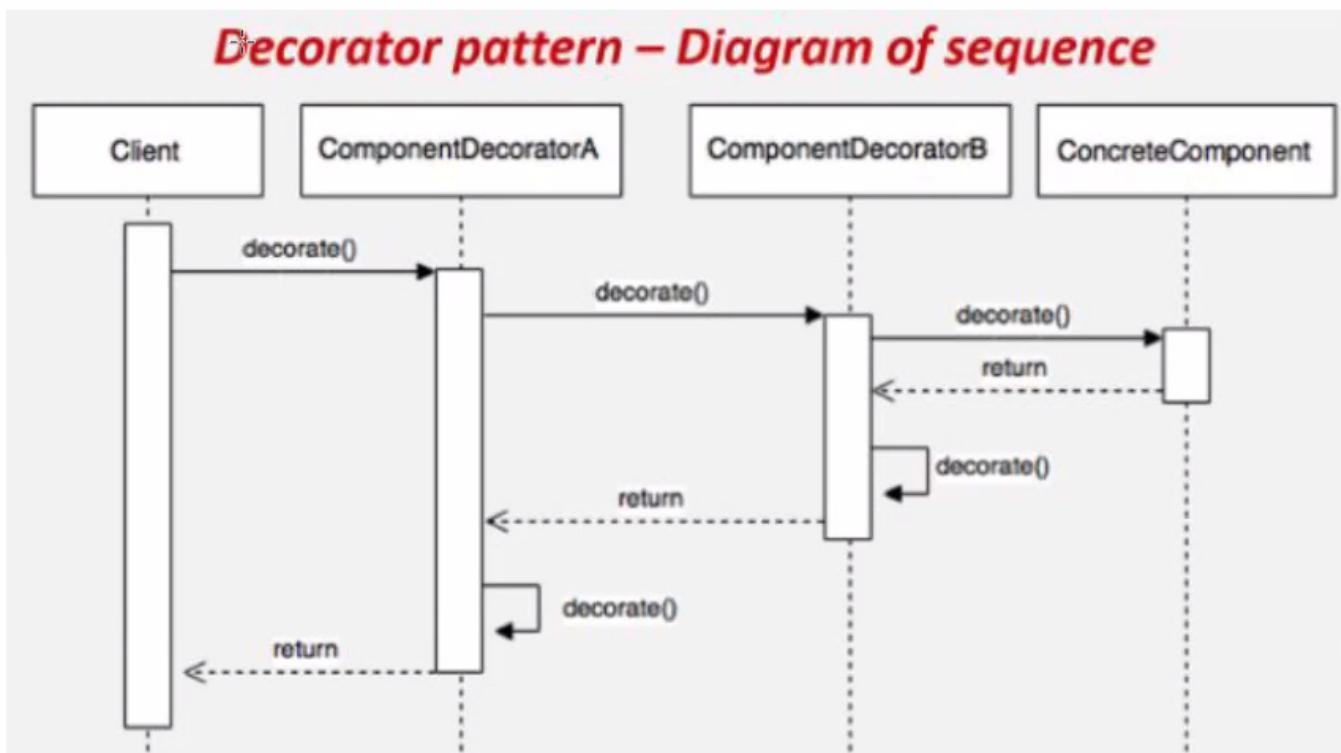
Propósito: Asigna responsabilidades adicionales a un objeto dinámicamente • Alternativa flexible a la herencia para extender la funcionalidad.

Motivación

- Añadir responsabilidades a obj. individual en vez de a una clase • Herencia no apropiada porque decora toda instancia igual
- Transparencia del decorador implementando interfaz componentE



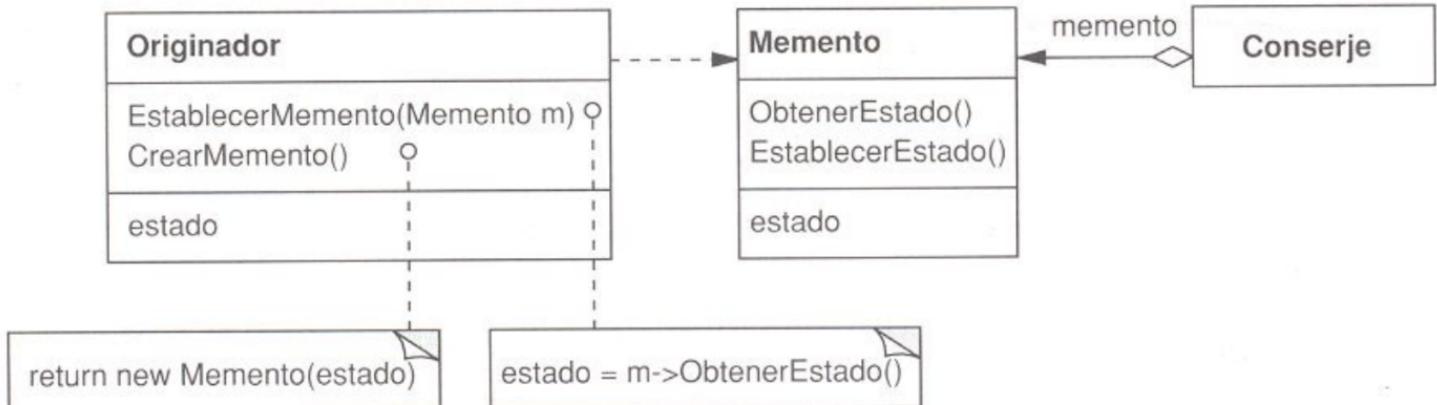
Secuencia



Patrón Memento

Propósito: Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación. Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado mas tarde.

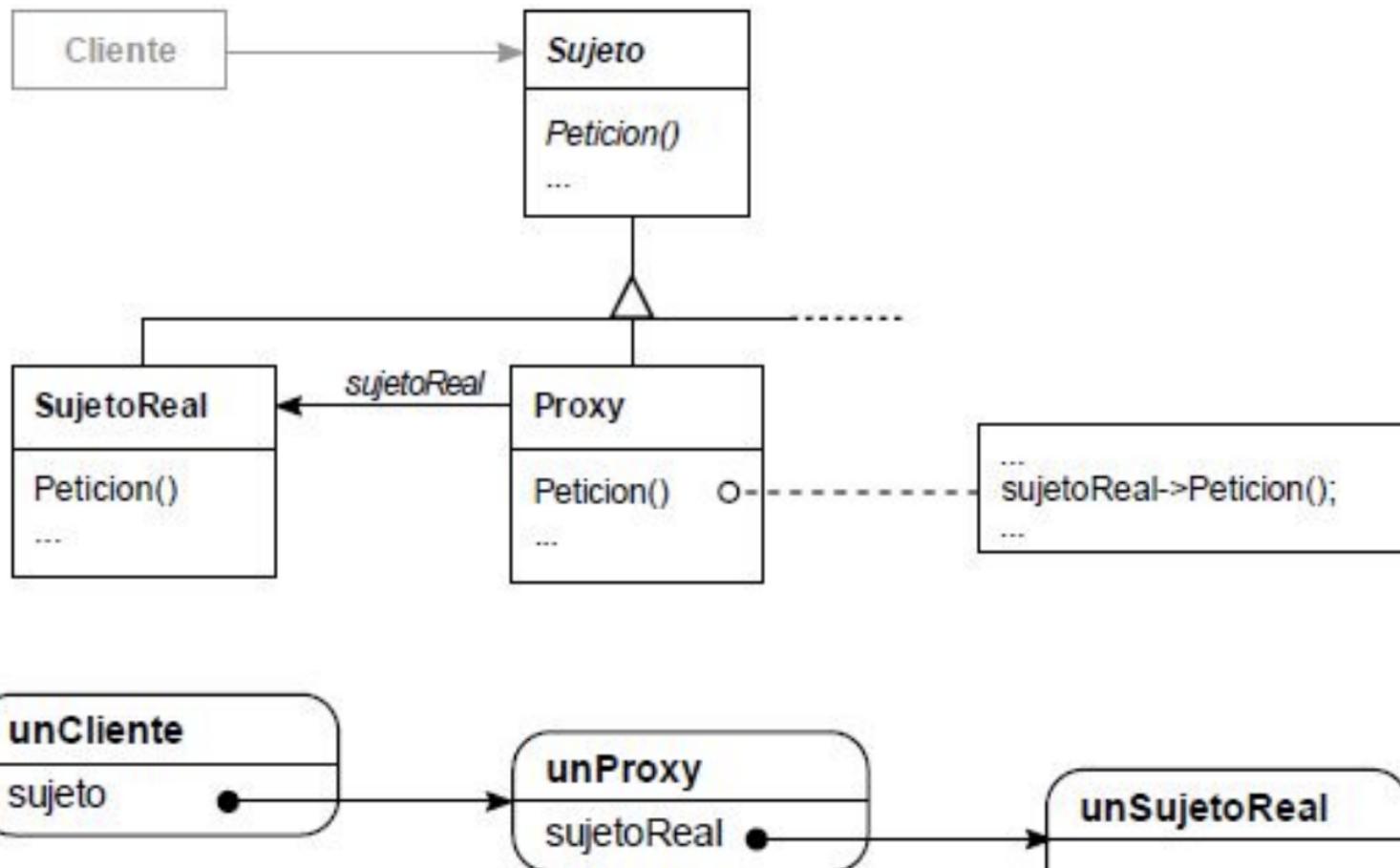
Motivación: Muchas veces es necesario almacenar el estado interno de un objeto, para por ejemplo implementar el deshacer.



Patrón Proxy

-Propósito: Permite proporcionar un sustituto o placeholder para otro objeto. Un proxy controla el acceso al objeto original, lo que le permite realizar algo antes o después de que la solicitud llegue al objeto original.

-Motivación: Una razón para controlar el acceso a un objeto es retrasar todo el coste de su creación e inicialización hasta que sea realmente necesario usarlo. Esto no debería influir en el cliente. La solución es utilizar otro objeto, un proxy, que actúa como un sustituto del real. El proxy se comporta igual que el objeto y se encarga de utilizarlo cuando sea necesario.



- Proxy

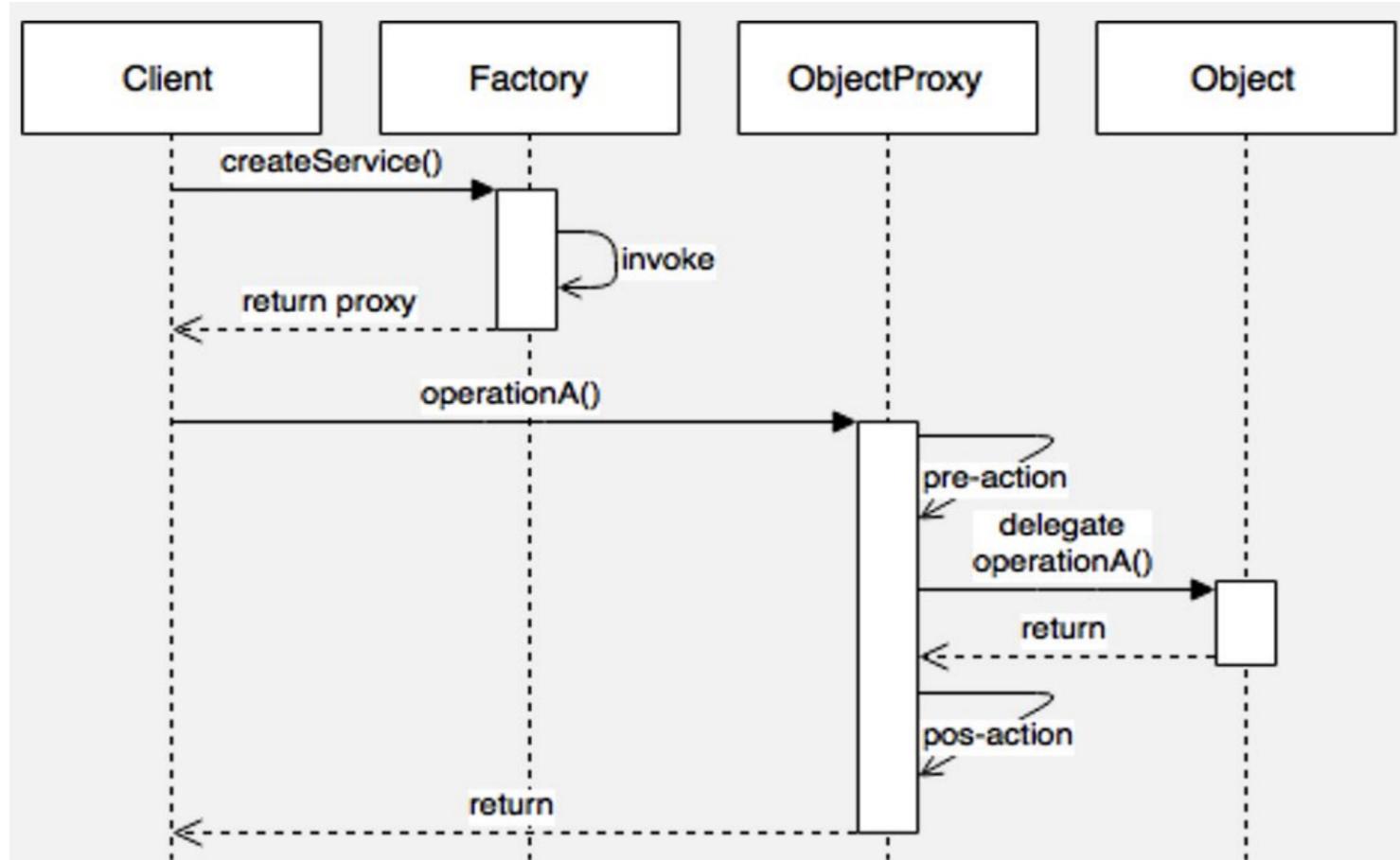
- mantiene referencia que permite acceder al objetoR.proxy refiere a un Sujeto en caso que las interfaces de SujetoReal y Sujeto sean la misma.
- proporciona una interfaz idéntica al Sujeto, ya que un proxy pueda ser sustituido por el sujeto real.
- controla el acceso al sujeto real, y puede ser responsable de su creación y borrado.
- otras responsabilidades dependen del tipo de proxy:
 - los proxies remotos son responsables de codificar una petición y sus argumentos para enviar la petición codificada al sujeto real que se encuentra en un espacio de direcciones diferente.
 - los proxies virtuales pueden guardar información adicional sobre el sujeto real, por lo que pueden retardar el acceso al mismo.
 - los proxies de protección comprueban que el llamador tenga los permisos de acceso necesarios para realizar una petición.

- Sujeto

- define la interfaz común para el SujetoReal y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un SujetoReal.

- SujetoReal

- define el objeto real representado.



Proxy Aplicabilidad

- Inicialización diferida (proxy virtual). Es cuando tienes un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesites de vez en cuando.
- Control de acceso (proxy de protección). Es cuando quieres que únicamente clientes específicos sean capaces de utilizar el objeto de servicio, por ejemplo, cuando tus objetos son partes fundamentales de un sistema operativo y los clientes son varias aplicaciones lanzadas (incluyendo maliciosas).
- Ejecución local de un servicio remoto (proxy remoto). Es cuando el objeto de servicio se ubica en un servidor remoto.
- Solicitudes de registro (proxy de registro). Es cuando quieres mantener un historial de solicitudes al objeto de servicio.
- Resultados de solicitudes en caché (proxy de caché). Es cuando necesitas guardar en caché resultados de solicitudes de clientes y gestionar el ciclo de vida de ese caché, especialmente si los resultados son muchos.
- Referencia inteligente. Es cuando debes ser capaz de desechar un objeto pesado una vez que no haya clientes que lo utilicen.

Preguntas parcial:

- Cual es la generalización
- Cual es la especialización
- Objeto
 - Es la representación de un ente del dominio del problema. Tienen un determinado estado interno y responsabilidades. Los objetos responden a los pedidos interactuando con los otros objetos que conoce.
- Abstracción
 - Clasificar los objetos, quitando las acciones y propiedades que no son necesarias.
- Clases
 - Representa un molde (template) para objetos de un mismo tipo.
- POO
 - Encapsulamiento: ocultamiento del estado interno de un objeto. Permite separar qué hacen los objetos (responsabilidad) de cómo lo hacen (implementación).
 - Herencia: jerarquía de clases.
 - Abstracción: expresa las características esenciales de un objeto, las cuales distinguen a un objeto de los demás.
 - Polimorfismo: permite al objeto emisor del mensaje despreocuparse de quién es exactamente su colaborador, sólo le interesa que sea responsable de llevar adelante la tarea que le encomienda a través del "mensaje".

CALENDARIO 2º CUAT. 2022

Carrera : ANALISTA de SISTEMAS
 CURSO : <<Curso / s>>
 PROFESOR : <<Nombre del profesor>>

N O C T U R N O						CLASES EFECTIVAS	
MES	Lun	Mar	Mié	Jue	Sem	<<Nombre de la materia>>	
Ago	8	9	10	11	1		
	15	16	17	18	2		
	22	23	24	25	3		
	29	30	31	1	4		
Set	5	6	7	8	5		
	12	13	14	15	6		
	19	20	21	22	7		
	26	27	28	29	8		
Oct	3	4	5	6	9		
	10	11	12	13	10		
	17	18	19	20	11		
	24	25	26	27	12		
Nov	31	1	2	3	13		
	7	8	9	10	14		
	14	15	16	17	15		
	21	22	23	24	16		
Dic	28	29	30	31		Exámenes Finales / Recuperatorios	
	5	6	7	8			
	12	13	14	15			
	19	20	21	22		RECESO (a confirmar)	
DN / NN	FERIADO DIURNO y NOCTURNO / NOCTURNO						
PC	PARCIAL						
FF	EXAMEN FINAL						
N	FERIADO						
T	SE INGRESA A LAS 19:40 HS						

Arquitectura de software

<https://www.youtube.com/watch?v=ILvYAzXO7Ek>

Organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principio que orientan su diseño y evolución

- Alude a la vista estructural general del software de alto nivel y el modo en que la estructura ofrece una integridad conceptual al sistema.
- Define estilo o combinación de estilos para una solución
- Se concentra en requerimientos no funcionales
- Esencial para éxito o fracaso de un proyecto

Tipologías (tipos de arquitectura)

- Empresarial
- Infraestructura (dentro de la empresarial)
- Software o de Aplicaciones
- Integración

¿PARA QUÉ SON LOS ESTILOS ARQUITECTÓNICOS?

- Definen los patrones posibles de las aplicaciones
- Permiten evaluar alternativas con ventajas y desventajas conocidas ante diferentes conjuntos de requerimientos no funcionales

- Sirven para sintetizar estructuras de soluciones

Algunos patrones de arquitectura

- Patrón de capas
- Modelo-vista-controlador
- Patrón cliente-servidor
- Patrón maestro-esclavo
- Patrón de filtro de tubería
- Patrón de intermediario
- Patrón de igual a igual
- Patrón de bus de evento
- Patrón de pizarra
- Patrón de intérprete
- Hexagonal
- Onion
- Clean

Arquitectura de capas

- Se definen un conjunto de niveles o capas, en la cual cada nivel interno que se atraviesa se aproxima mas al nivel del conjunto de instrucciones de máquina.
- Sistemas en capas puros: cada capa solo puede comunicarse con la vecina. Aunque esta solución aunque puede ser menos eficiente en algunos casos facilita la portabilidad de los diseños.

- Capa de presentación
 - Presentación de interfaz de usuario
 - Captura de datos del usuario
 - Validación de datos del usuario
- Capa de negocio
 - Modelo de negocio
 - Recepción de solicitudes de la capa de presentación
 - Llamada a la capa de datos
- Capa de datos
 - Almacenamiento de datos.
 - Recuperación de datos
 - Interacción con la capa de negocio

Trabajo práctico

- Identificar cada patrón

- 40 clases mínimas

-

proxy

memento

state

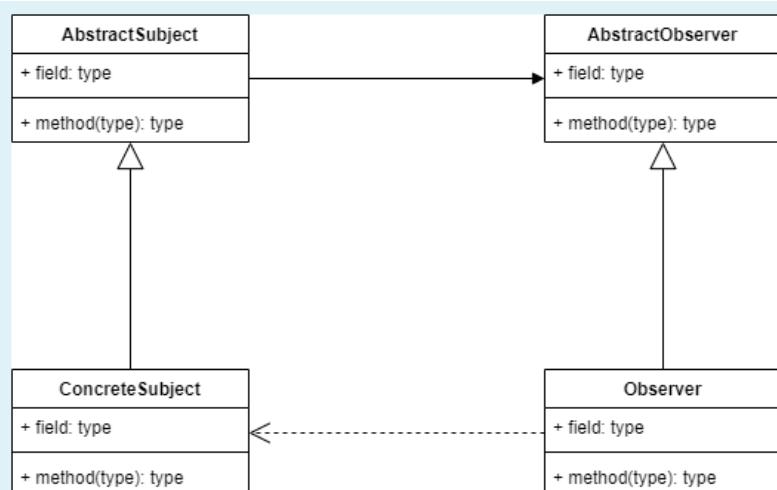
observa

strategy
singleton
composite
factory

#memento#

Secuencia → 4 diagramas de secuencia, hacer flujos mas complejos que otros, mejor vistosos

- responsabilidades
- flechas
- herencia hacia arriba
- responsabilidades
- si está bien implementado, si esta bien el patrón



En un observer

Seleccione una:

- a. La relación entre AbstractSubject y ConcreteSubject puede ser una implementación de interfaces
- b. No hay relación entre AbstractSubject y AbstractObserver
- c. La relación entre ConcreteSubject y ConcreteObserver tiene que ser de uso
- d. La clase AbstractSubject debe tener implementado los métodos de Add, Remove y Notify
- e. La clase AbstractObserver debe tener implementado los métodos de Add, Remove y Notify
- f. La clase AbstractObserver debe tener implementado el método Update
- g. La clase ConcreteObserver debe tener implementado el método Update
- h. A, B, C y F son correctas
- i. B, D y F son correctas
- j. D y G son correctas
- k. Ninguna es correcta