Diseño de Aplicaciones 2

Resumen

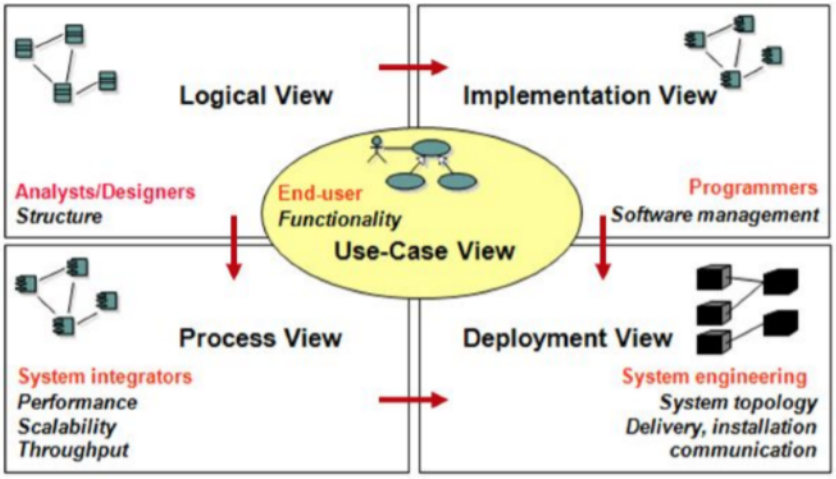
Contenido

Introducción a la arquitectura, modelo 4 + 1, UML Avanzado

Concepto de arquitectura y descripción del modelo 4+1

Es un modelo para describir la arquitectura de sistemas de software basado en el uso de diferentes perspectivas concurrentes.

Las vistas describen el sistema desde el punto de vista de diferentes interesados (usuarios, desarrolladores, directores de proyecto, etc). El modelo cuenta con 4 vistas bien diferenciadas y estas 4 vistas se han de relacionar entre sí con una vista más, que es la denominada vista "+1".



Vista Lógica

Describe el modelo desde los elementos de diseño y sus interacciones. En esta vista se representa la funcionalidad que el sistema proporcionará a los *usuarios finales*. Es decir, se ha de representar lo que el sistema debe hacer, y las funciones y servicios que ofrece.

Aspectos estáticos

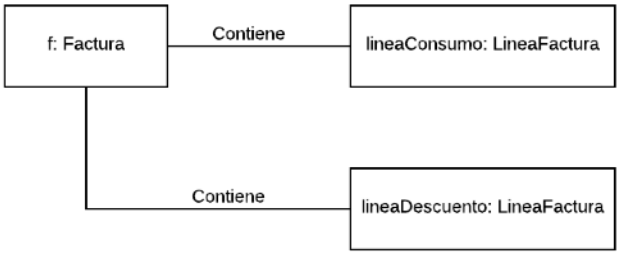
- Diagramas de clases
- Diagramas de paquetes

Aspectos dinámicos

- Diagramas de secuencia
- Diagramas de comunicación

Diagramas de Objetos

Notacion: Muy simple
Son útiles para mostrar cómo se relacionan los objetos en un escenario particular.



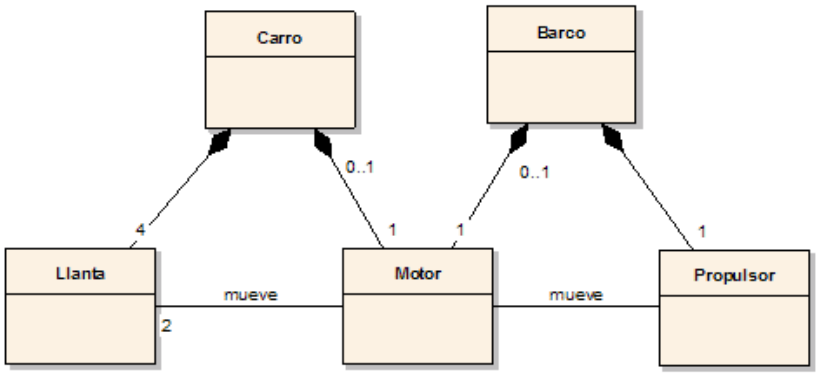
Diagramas de estructura compuesta

Un **diagrama de estructura** es un tipo de diagrama en el Lenguaje de Modelado Unificado (UML), que muestra la estructura interna de una clase y las *colaboraciones* que esta estructura hace posibles. Esto puede incluir *partes* internas, *puertos* mediante las cuales las partes

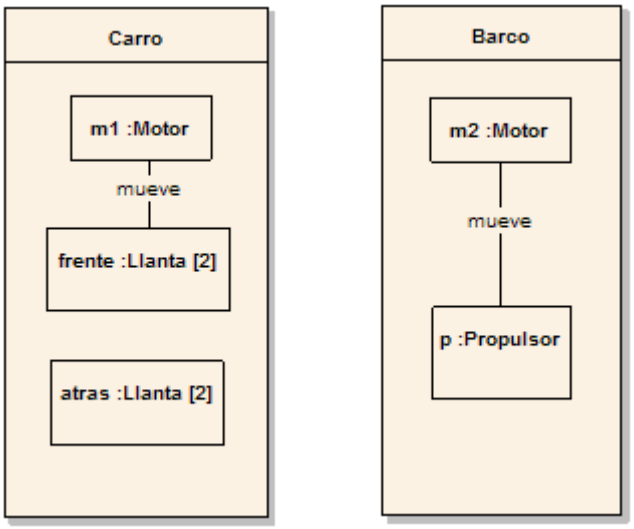
interactúan con cada una de las otras partes, instancias de las clases que interactúan con las partes y con el mundo exterior, y *conectores* entre partes o puertas.

Una *estructura compuesta* es un conjunto de elementos interconectados que colaboran en tiempo de ejecución para lograr algún propósito. Cada elemento tiene algún *rol* definido en la colaboración.

ANTES (Diagrama de clases)



AHORA (Diagrama de estructura compuesta)



Va en línea punteada si no es relación de composición

Elementos UML 2

- Clase: Para mostrar la parte de la cual se ilustra su composición interna (ejemplo: carro o Barco)
- Parte: Se muestra con un rectángulo, e indica los objetos que conforman al objeto principal. Ejemplo: el motor y las llantas en el carro, o el motor y el propulsor en el Barco. Si se coloca una parte dentro de una clase significa, en un diagrama de clases, que la clase contenedor tiene una relación de composición con dicho elemento.
- Conector:Indica la relación entre las parte internas de la clase que se analiza.
- Puertos: Se pueden mostrar puertos para indicar la entrada o salida de una parte hacia otra parte. Se muestran como pequeños cuadrados al final de un conector entre dos partes. No son obligatorias, pero son recomendables si se quiere encapsular el funcionamiento de las partes.

Diagramas de Estado

Intención: Permitir a un objeto modificar su comportamiento cuando su estado interno cambia.

Maquina de estados

Una máquina de estados es un comportamiento que especifica las secuencias de estados por las que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus respuestas a esos eventos

Estado: Es una condición o situación en la vida de un objeto en la que satisface una condición, realiza una actividad o espera por algún evento. Cada estado es único para la máquina. Tiene acciones instantáneas y no interrumpibles que se realizan al entrar al estado (entry) y al salir de él (exit). Cuando un objeto está en un estado puede realizar una actividad prolongada o interrumpible (do). Un estado puede tener transiciones internas que permiten manejar eventos sin salir del estado (on event).

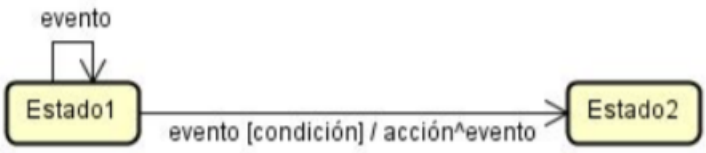
Evento: Es la especificación de un estímulo que tiene lugar en el tiempo y espacio y que puede disparar una transición entre estados. Pueden ser internos (entre objetos que pertenecen al sistema) o externos (entre actores y el sistema).

Eventos de tiempo: representan el pasaje de un período de tiempo. Se indican con la palabra clave after(expresión).
Eventos de cambio: son eventos que representan el cambio en un estado o la satisfacción de alguna expresión when(expresión booleana). Se puede usar [___].

Transición: Representa el cambio ante un evento de un estado a un estado sucesor, o a sí mismo. Cada transición es única para un estado.
Existen dos tipos de transiciones:

- No automática: generada por un evento.
- Automática: que no tiene un evento asociado.

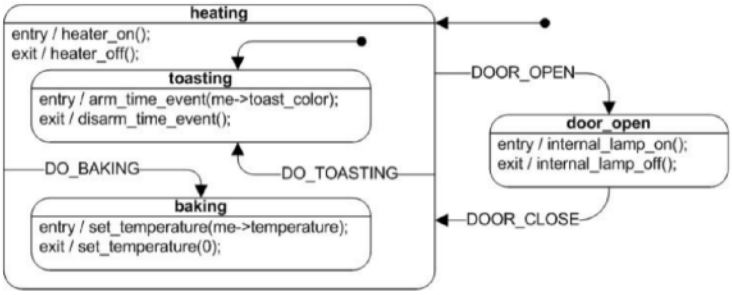
Ante un evento desde un estado solo se puede disparar una transición



Subestados
Un estado puede tener varios niveles de profundidad. Un estado padre (superiores) y estados internos. Las transiciones pueden comenzar y terminar en cualquier nivel.

Reglas

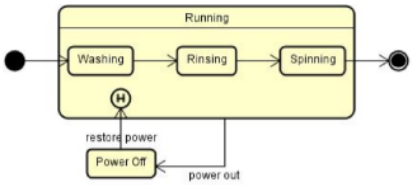
- Una transición puede llegar al estado compuesto o a un subestado. Si la transición llega a un subestado la ejecución comienza en él. Si llega al compuesto, el mismo debe incluir un estado inicial para comenzar la ejecución de la máquina. Las acciones entry se ejecutan siempre topdown, desde afuera hacia adentro, primero las entry del compuesto y luego la de los subestados).
- Una transición puede salir de un estado compuesto o directamente de un subestado. Las acciones exit se ejecutan bottomup, desde adentro hacia afuera, primero las exit del estado de donde sale y luego la del compuesto.
- Una transición desde el estado compuesto implica que se puede salir por esa transición desde cualquier subestado.



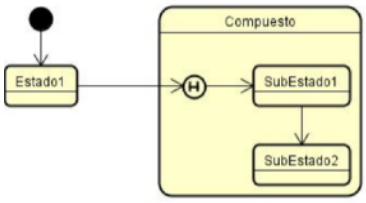
Historia

Es posible en un diagrama marcar o recordar el último subestado activo antes de salir del estado compuesto.

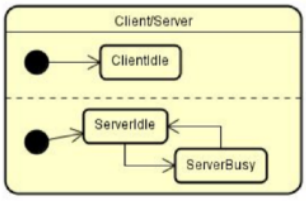
- Shallow history (H): retoma en el último subestado en el que se encontró
- Deep history (H*): retoma en el último subestado del último subestado en el que se encontró



Si el compuesto tiene un punto de entrada se puede marcar la historia así también:

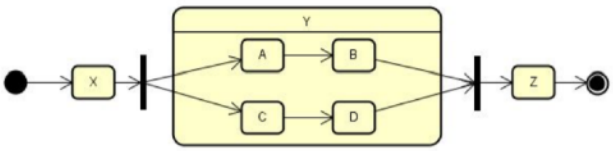


Regiones ortogonales



Ambas regiones se encuentran activas si el estado Client/Server está activo

Forks y Joins



Opciones de implementación:

- Switch
- State pattern
- Tablas

Vista de Procesos

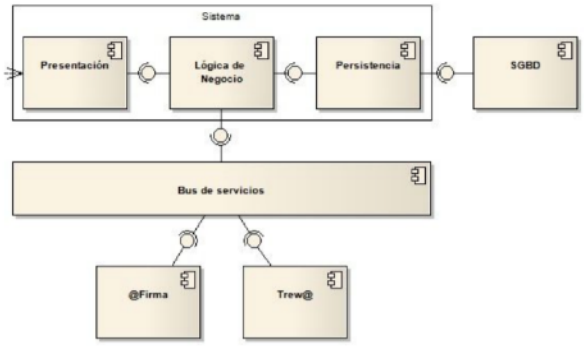
En esta vista se muestran los procesos que hay en el sistema y la forma en la que se comunican estos procesos; es decir, se representa desde la perspectiva de un *integrador de sistemas*, el flujo de trabajo paso a paso de negocio y el flujo operacional de los componentes que conforman el sistema. Para completar la documentación de esta vista se puede incluir el diagrama de actividad de UML.

Vista de Implementación

En esta vista se muestra el sistema desde la perspectiva de *un programador* y se ocupa de la gestión del software; o en otras palabras, se va a mostrar cómo está dividido el sistema software en componentes y las dependencias que hay entre esos componentes. Para completar la documentación de esta vista se pueden incluir los diagramas de componentes y de paquetes de UML.

Diagramas de Componentes

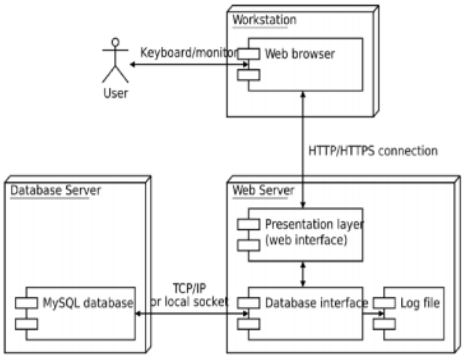
Un diagrama de componentes representa cómo un sistema de software es dividido en componentes y muestra las dependencias entre estos componentes. Los componentes físicos incluyen archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables, o paquetes. Debido a que los diagramas de componentes son más parecidos a los diagramas de casos de usos, éstos son utilizados para modelar la vista estática y dinámica de un sistema. Muestra la organización y las dependencias entre un conjunto de componentes.



Vista de Entrega (Deployment)

En esta vista se muestra desde la perspectiva de un ingeniero de sistemas todos los componentes físicos del sistema así como las conexiones físicas entre esos componentes que conforman la solución (incluyendo los servicios). Para completar la documentación de esta vista se puede incluir el diagrama de despliegue de UML.

Diagramas de Entrega.



Vista de casos de uso (+1):

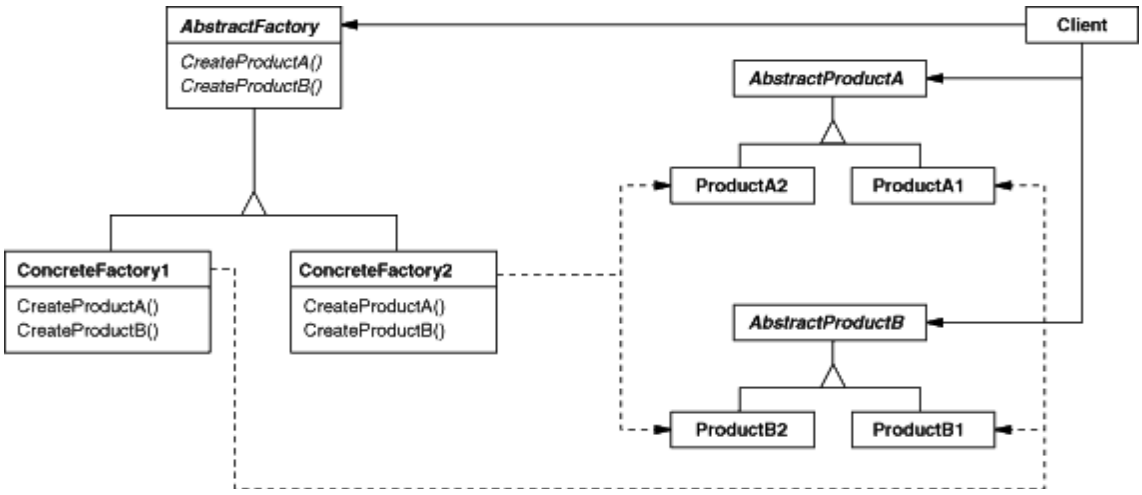
Esta vista va a ser representada por los casos de uso software y va a tener la función de unir y relacionar las otras 4 vistas, esto quiere decir que desde un caso de uso podemos ver cómo se van ligando las otras 4 vistas, con lo que tendremos una trazabilidad de componentes, clases, equipos, paquetes, etc, para realizar cada caso de uso. Para completar la documentación de esta vista se pueden incluir el diagramas de casos de uso de UML.

Aplicación de patrones de diseño

Creacionales

Abstract Factory:

Motivación: Proveer una interfaz para la creación de familias de objetos relacionados o dependientes sin especificar las clases concretas.

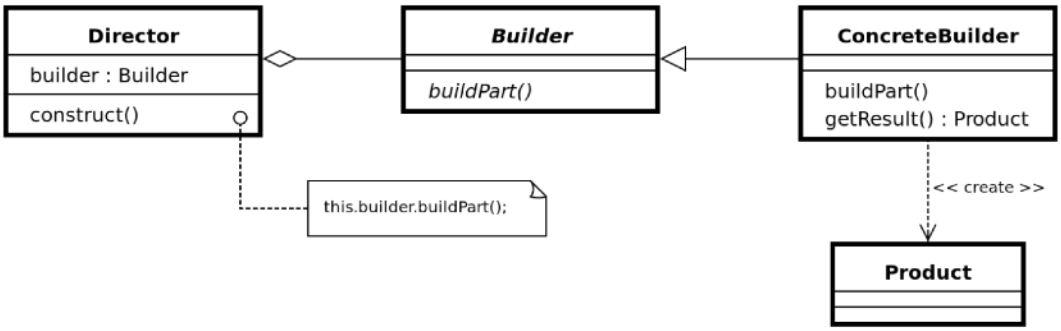


Builder

Es un patrón creacional.
La idea del builder es setear los atributos en la inicialización que uno considere necesaria.
Intención: Separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

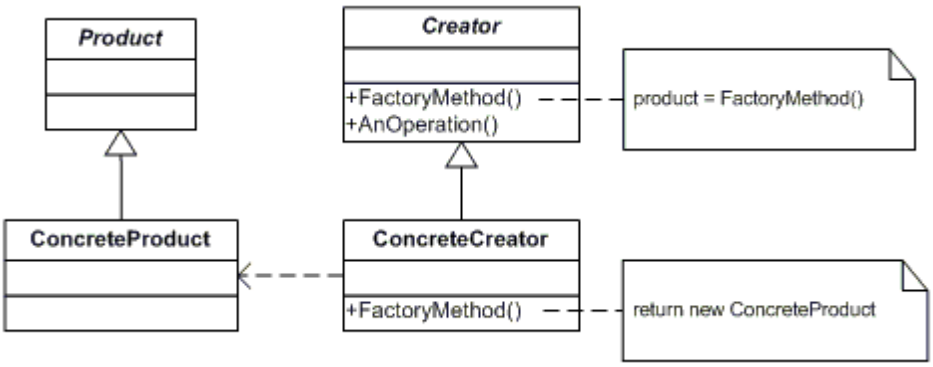
Estructura:

- El producto: Representa el objeto complejo que se quiere construir.
- El Builder: Especifica una interface para crear las partes del objeto producto.
- ConcreteBuilder: Construye y ensambla las partes del producto mediante la implementación de la interface builder. Define y controla la representación que crea. Provee una interface para obtener el producto.
- Director: Construye un objeto utilizando la interface del builder.



Factory Method:

Motivación: Definir una interfaz para la creación de un objeto, pero delegar la responsabilidad de la creación a las subclases, las cuales deciden qué clase instanciar.

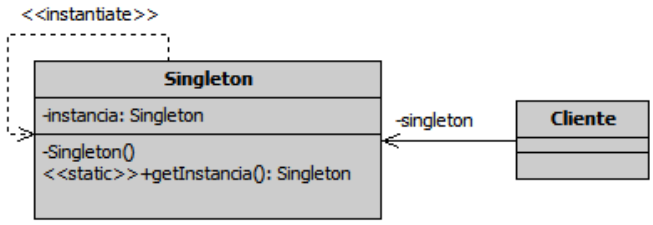


Aplicación:

- Se utiliza cuando una clase no puede anticipar qué clase debe crear.
- Cuando una clase quiere que sus subclases sean las responsables de crear la instancia.

Singleton

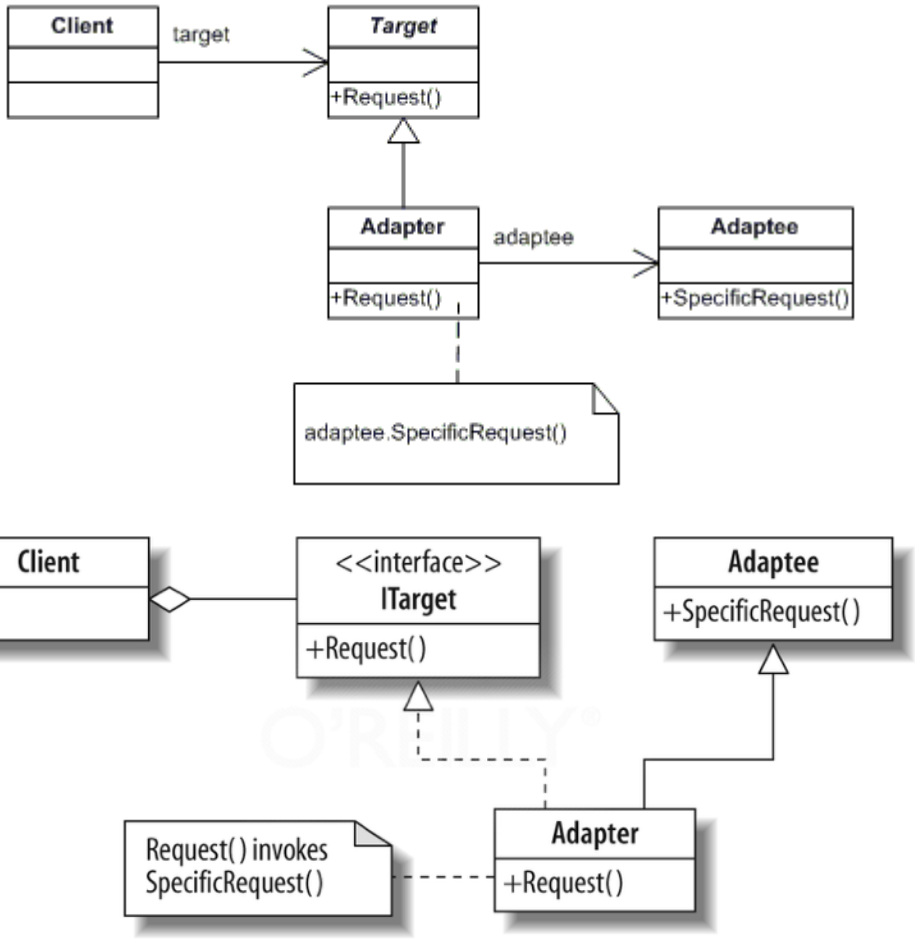
Es un patrón creacional.
Intención: Una clase de la cual puede existir una sola instancia.
Aplicabilidad: La clave de aplicar un singleton es prevenir a los programadores, clientes de tener cualquier acceso a la creación de un objeto, excepto la forma que nosotros proveemos, es decir acceso único al objeto.
Estructura: Clase singleton con constructor privado, atributo estático de la clase y un método GetInstance():Singleton



Estructurales

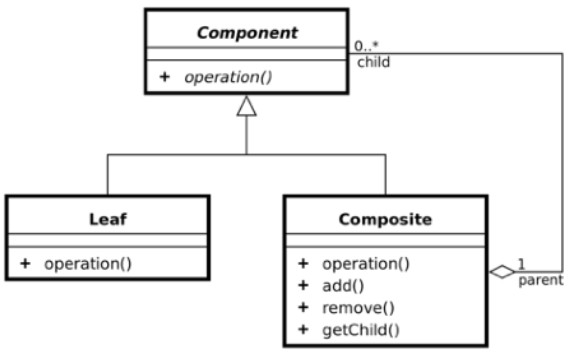
Adapter Pattern:

Motivación: Convertir la interfaz de una clase en otra interfaz que los clientes esperan. El adaptador permite a un conjunto de clases trabajar juntas que de otra forma sería imposible por compatibilidad de interfaces.



Composite:

Motivación: Componer objetos en una estructura arborescente para representar una jerarquía parcial. Este patrón permite a los clientes tratar a los objetos individuales y a los compuestos como una misma cosa.



Aplicación:

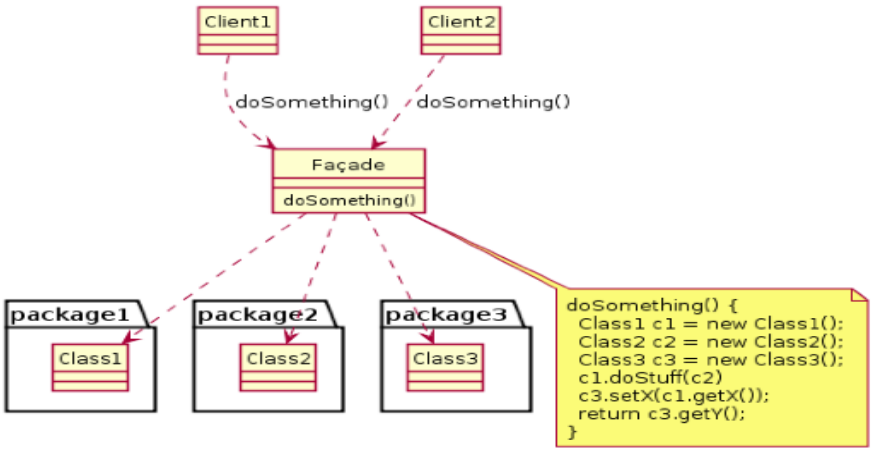
- Cuando se quiere representar una jerarquía parcial.
- Cuando se quiere que los clientes ignoren las diferencias entre objetos compuestos y objetos atómicos.

Ventajas:

- Permite tratar los componentes de igual forma independientemente de la complejidad de los mismos.
- Permite agregar nuevos componentes sin cambios mayores.

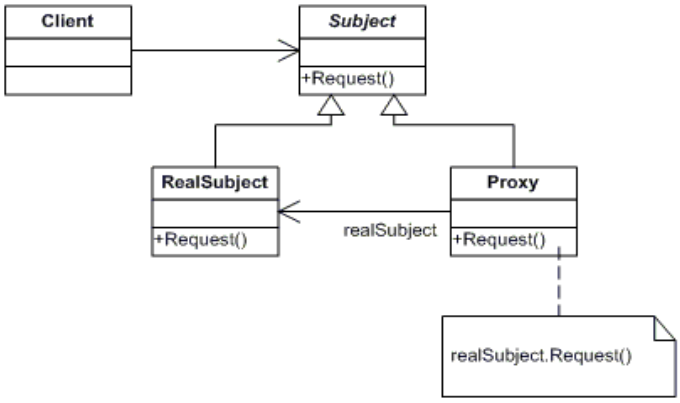
Facade

Es un patrón estructural.
Intención: Proveer una interfaz unificada para un conjunto de interfaces en un subsistema. La fachada define una interface que hace fácil de usar los subsistemas.
Motivación: Estructurar un sistema en subsistemas ayuda a reducir la complejidad y el número de dependencias.



Proxy

Motivación: Permitir que los clientes de un componente, se comuniquen con un representante del mismo, en lugar de con el objeto. De forma de tener control sobre el acceso al objeto. Siempre que sea necesario, tener una referencia controlada al objeto.



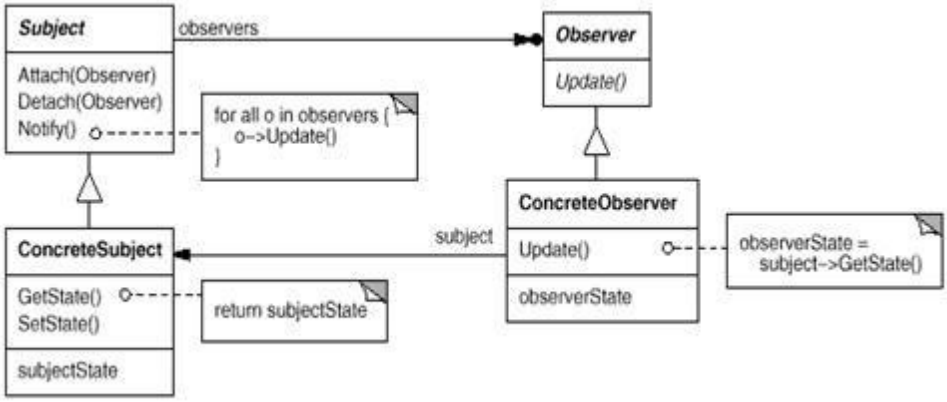
Aplicación:

- Proxy remoto: Es un representante local para un objeto en diferentes espacios o máquinas.
- Proxy virtual: Es un representante más "liviano" que el objeto original, el cual es creado a demanda.
- Proxy de protección: Es un representante que controla el acceso al objeto original. Es útil cuando un objeto debería tener distintos permisos de acceso.
- Referencia inteligente: Es un puntero inteligente que realiza operaciones extra, como:
    - Contar cuantos lo apuntan.
    - Crear una instancia del objeto real en memoria cuando es solicitado por primera vez.
    - Verificar el acceso concurrente a un objeto.

Comportamiento

Observer:

Motivación: Definir una relación de uno a muchos entre objetos de tal manera que cuando uno cambia de estado, todos los demás dependientes de él son notificados automáticamente.



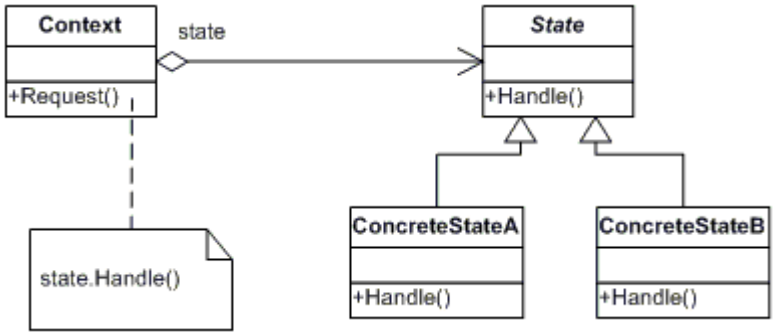
Aplicación:

- Cuando una abstracción tiene dos aspectos, dependientes entre ellos. Encapsular estos aspectos en objetos separados permite modificarlos independientemente.
- Cuando el cambio en un objeto provoca cambios en otros objetos y no es conocida la cantidad de objetos que van a cambiar.
- Cuando un objeto debería ser capaz de notificar a otros objetos sin necesidad de saber cuales son estos objetos. Así evitando acoplamiento con esos objetos.

Ventajas:
Se pueden combinar o agregar los observadores en forma independiente, cada observador da una visión independiente del objeto.
Desventajas:
Updates que sirven para un observador no sirven para todos.

State Pattern:

Motivación: Permitir un objeto cambiar su comportamiento en base a su estado.
Uso: Un objeto debe cambiar de comportamiento en tiempo de ejecución.
Una operación muy compleja con condiciones y posibles valores predeterminados, y que se evalúa la condición en varios lugares para determinar el comportamiento.



Notas:
El cambio de estado puede estar en el contexto o en cada uno de los estados concretos.
Los objetos estados pueden compartirse si no tienen variables de instancia.

Ventajas:

- Localiza y divide conductas de los distintos estados, poniendo todo lo relacionado a un estado en un solo lugar.
- Es una buena alternativa frente a opciones case o if que se repiten en el contexto.
- Es muy sencillo agregar nuevos estados.
- La transición de los estados es explícita y atómica desde el punto de vista del contexto.

Desventajas:

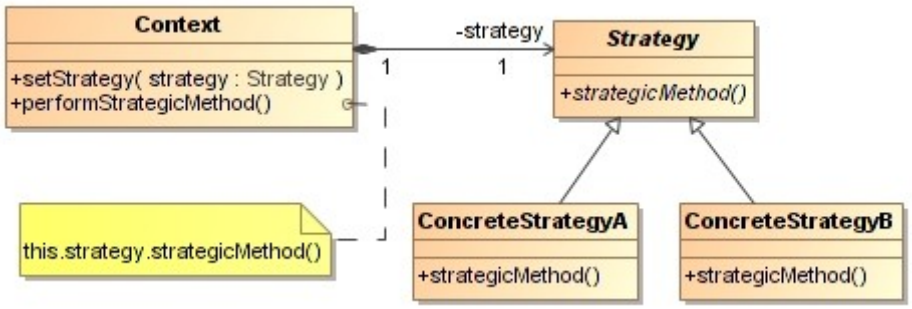
- Aumenta el número de clases y la solución es menos compacta que con una sola clase.
- Quién realiza las transiciones? El patrón no lo define. Si son simples, pueden ser definidas en el contexto. Definidas en los estados son más flexibles. Otra forma es utilizando tablas para guardar las transiciones.

Maneras de crear los estados:

- Cada vez que se necesitan: Cuando no se conocen los estados ni cuando estos cambian.
- Todos al crear el contexto: Cuando hay muchos y rápidos cambios. El contexto debe tener referencias a todos los estados posibles.

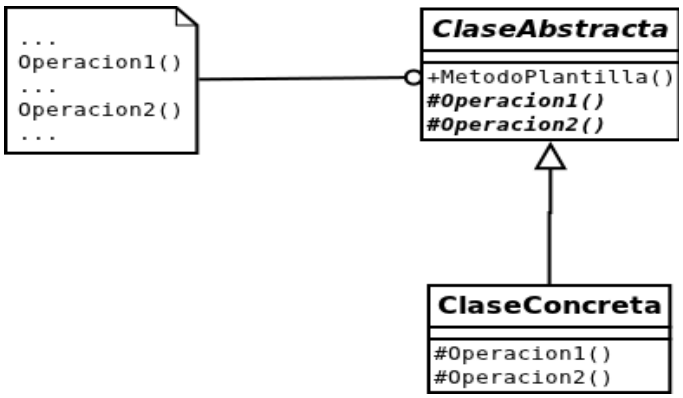
Strategy

Es un patrón de comportamiento.
Intención: Definir una familia de algoritmos, encapsular cada uno en una clase y usarlos de forma intercambiable. Permite intercambiar el algoritmo independientemente de los clientes que lo usen.
Aplicabilidad: Se necesitan diferentes variantes de un algoritmo. Una clase define variantes de su comportamiento.
Estructura:



Template Method

Es un patrón de comportamiento.
Intención: Definir el esqueleto de un algoritmo de una operación mientras se difieren algunos pasos a la subclase.
Aplicabilidad: Cuando una **porción** de un algoritmo a ser usado en una clase es dependiente del tipo de objeto de las subclases.
Beneficios:
Gran flexibilidad y reuso
Fácil de modificar algoritmos
Implementar de una sola vez las partes invariantes de un algoritmo dejando las partes que varían a una subclase



Principios de Diseño

Principios SOLID

Los SOLID representan cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar con el tiempo.

Single Responsibility Principle (SRP)

Una clase solo debe tener una razón por la cual cambiar. Solo debe capturar una abstracción de la realidad y sus responsabilidades.

Open/Closed Principle (OCP)

Las entidades de software deben ser abiertas (Open) para la extensión y cerradas (Closed) para la modificación. Abierto es poder cambiar el comportamiento de la clase ante cambios en la aplicación o para satisfacer nuevos requerimientos y cerrado quiere decir que el código de la clase no se debe cambiar ya que otras clases podrían estar utilizándolo.

Liskov Substitution principle (LSP)

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. Toda subclase debe ser sustituible por su clase base.

Interface Segregation Principle (ISP)

No debe de existir una interfaz “pesada” sino que debemos poder separar esta gran interfaz en varias más pequeñas para así las clases poseer aquellos métodos que realmente quieran y no otros.

Dependency Inversion Principle (DIP)

Los módulos que implementan funcionalidad de alto nivel no deben depender de módulos que implementan la funcionalidad de bajo nivel. Ambos deben depender de interfaces bien definidas.

En todos los casos, es necesario comprender el enunciado de cada principio, y ser capaz de identificar si se cumple o no en una situación dada, pudiendo justificar la respuesta y proponer cambios para mejorar el diseño en base a ellos.

Principios de paquetes (Cohesión)

Principio equivalencia de reúso/liberación (REP)

Principio de Reuso

- La idea principal es liberar lo que se quiera reusar (aplicar reúso).
- Separo lo que voy a reusar de lo que no, y lo libero en un ensamblado.

En .NET la unidad de release es el assembly, entonces la unidad de reuso también lo es.

Principio de clausura común (CCP)

Principio de mantenimiento

Las clases pertenecientes a un paquete deben de cambiar por el mismo tipo de cambio.
Si un cambio afecta a un paquete, entonces afecta a todos las clases del mismo y a ningún otro paquete.
Clases que cambian juntas, pertenecen juntas.
Clases candidatas a cambiar por un mismo motivo se deben agrupar en un mismo paquete, de forma de minimizar el impacto del cambio.

Principio de reúso común (CRP)

Principio de Reuso

- Las clases que pertenecen a un paquete se reusan juntas.
- Poner todo lo que se reuse en un mismo paquete.

Se recomienda favorecer primero al CCP (al comienzo del diseño) y luego cuando la arquitectura se estabiliza favorecer a los otros dos.

Principios de paquetes (Acoplamiento)

Principio de dependencias acíclicas (ADP)

El grafo de dependencias entre paquetes debe ser dirigido y acíclico.
Como solucionar si se crea un ciclo:

- Con inversión de dependencias (DIP)
- Crear un nuevo paquete que agrupe clases que ambas dependen.

Principio de dependencias estables

- Las dependencias entre paquetes deben ir en el sentido de la estabilidad.
- Un paquete debe depender solamente de paquetes que sean mas estables que él.

Principio de abstracciones estables (SAP)

Un paquete debe ser tan abstracto como es su estabilidad.
Los paquetes más estables (dependen muchos de él y él no depende) deben tender a ser abstractos (con interfaces o clases abstractas).
Paquetes inestables deben ser concretos.
Un paquete estable debe ser abierto a la extensión y paquetes inestables deben ser fácilmente modificables. Por ejemplo mediante OCP un paquete estable puede ser fácilmente extensible.
Con DIP a nivel de paquete hago los paquetes estables abstractos y fácilmente extensibles por paquetes menos estables.

Métricas

Cohesión Relacional:

Indica cuán fuertemente relacionadas están las clases dentro de un paquete. Debería estar entre 1,5 y 4.

$$H = \frac{R + 1}{N}$$

R: Número de relaciones entre clases dentro de un paquete. Solo se cuentan las relaciones de asociación, composición y agregación. Las relaciones bidireccionales se cuentan por dos.
N: Número de clases e interfaces en un paquete.

Inestabilidad

Indica cuán inestable es un paquete. Refiriéndose a cuán fácil es cambiarlo dependiendo de cuántos paquetes lo utilizan.

$$I = \frac{C_e}{C_e + C_a}$$

Ca: Dependencias entrantes.
Ce: Dependencia salientes.

Abstracción:

Indica cuán abstracto es un paquete.

$$A = \frac{N_a}{N_c}$$

Na: Cantidad de clases abstractas e interfaces.
Nc: Cantidad de clases, incluyendo las abstractas e interfaces.