

Tema 6. Patrones de diseño.

Introducción.

Durante el diseño Orientado a Objetos es frecuente encontrarse repetidamente con ciertos tipos de problemas, para analizar, compartir y documentar el conocimiento sobre dichos tipos de problemas se han desarrollado los patrones de diseño.

Dichos patrones también se aplican en otras partes del desarrollo (análisis) y es recomendable usarlos en cualquier momento para aprovechar el conocimiento y la experiencia existente y no partir de cero.

Los patrones constituyen la solución base ideal o por lo menos la más recomendable a un problema concreto.

Definición.

Un patrón es un **conjunto de información** que proporciona respuesta a un conjunto de problemas similares. Para ello se aíslan los aspectos comunes y su solución y se añaden cuantos comentarios y ejemplos sean oportunos.

Podríamos definir patrón como una **solución a un problema en un contexto**.

- **Contexto.** Las situaciones recurrentes.
- **Problema.** Metas y restricciones en el contexto.
- **Solución.** Diseño para conseguir las metas dentro de las restricciones.

Los patrones ayudan a **capturar conocimiento** y a crear un vocabulario técnico, hacen el **diseño** orientado a objetos más **flexible, elegante** y en algunos casos **reusable**.

Normalmente un patrón se compone de los siguientes elementos:

- **Nombre.** Una o dos palabras, alto nivel de abstracción para facilitar la asimilación del funcionamiento del mismo. Estos nombres se incorporan a la memoria y son utilizados en la comunicación.
- **Problema.** Describe cuando se debe aplicar el patrón y su contexto.
- **Solución.** Detalla el mejor diseño disponible.
- **Consecuencias.** Describe las ventajas y desventajas del mismo.

Los patrones de diseño pueden relacionarse y aplicarse de forma simultánea en muchos casos.

Patrones GRASP.

General Responsibility Assignment Software Patterns (**GRASP**) nos da unos **principios generales** para asignar responsabilidades y se utiliza sobre todo en la realización de diagramas de interacción.

Una vez que tenemos el Modelo de dominio hay que decidir qué métodos añadir y dónde (parte esencial del desarrollo OO)

Básicamente las responsabilidades de un objeto son conocer y hacer. Dichas responsabilidades implicarán más o menos métodos y clases.

Principales patrones GRASP:

1. **Experto** (en Información).
2. **Creador**.
3. **Alta Cohesión**.
4. **Bajo Acoplamiento**.
5. **Controlador**.

Experto.

Problema:

¿Cuál es el principio más básico para añadir responsabilidades en una clase?

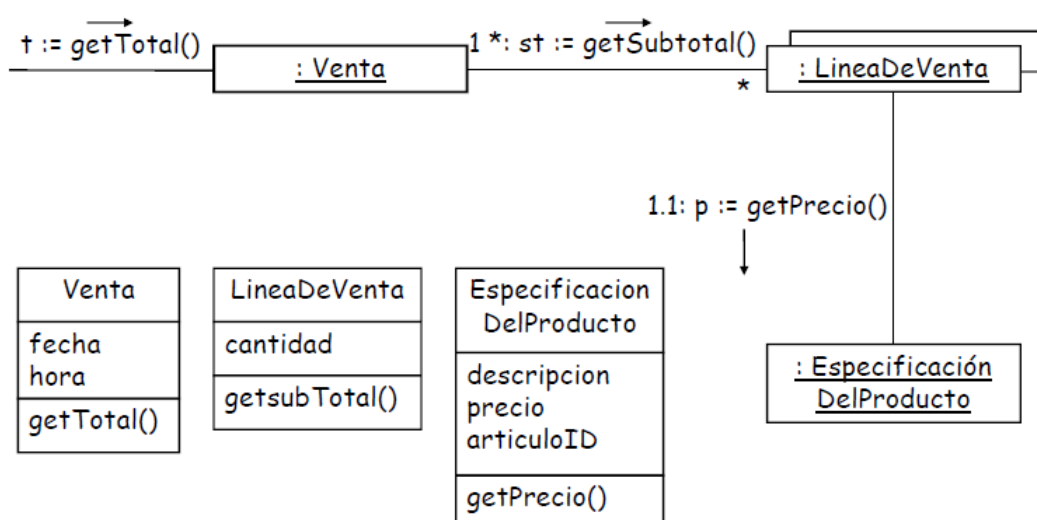
Solución:

Asignar responsabilidades al experto de la información, es decir, a la clase que tiene la información necesaria para llevar la tarea a cabo.

Consecuencias:

- Encapsulamiento de la información y, por ende, el bajo acoplamiento.
- El comportamiento distribuido entre las clases, es decir, clases más cohesivas.

Ejemplo:



Contraindicaciones:

En algunas ocasiones no es una guía deseable sobre todo por problemas de cohesión, acoplamiento y duplicación de código. (Si venta debe ser almacenada en una base de datos, el Experto sugiere a Venta como responsable, pues contiene mucha de la información a almacenar.

Patrones relacionados:

Bajo Acoplamiento y Alta Cohesión.

Creador.

Problema:

¿Quién debe ser responsable en la creación de una nueva instancia de una clase?

Solución:

Una clase B tiene la responsabilidad para crear una instancia de la clase A si:

- B agrega objetos de A.
- B contiene objetos de A.
- B almacena objetos de A.
- B usa objetos de A.
- B tiene los datos necesarios para inicializar a A cuando este es creado.

Consecuencias:

Soporta el bajo acoplamiento, es decir, este no se incrementa.

Ejemplo:

En la aplicación PDV, ¿quién debería crear una LineaDeVenta? Venta de hecho contiene (agrega) muchos objetos de LineaDeVenta, debe tener, por lo tanto, la responsabilidad de la creación de instancias de esta clase y por lo tanto contener un método crearLineaDeVenta.

Contraindicaciones:

A menudo la creación de instancias es una tarea compleja, en estos casos es aconsejable delegar la creación a una clase auxiliar denominada **Factoria**.

Patrones relacionados:

Bajo Acoplamiento y Factoría.

Alta Cohesión.

Problema:

¿Cómo lograr que la complejidad sea lo más manejable posible? Alta cohesión: lo relacionadas que están las responsabilidades de una clase, o una clase con responsabilidades altamente relacionadas y que no lleva a cabo gran cantidad de trabajo.

Solución:

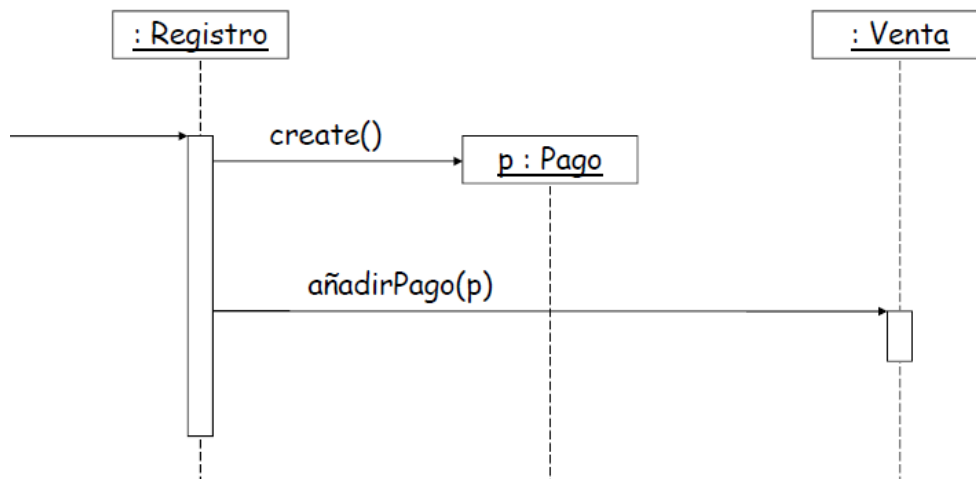
Asignar responsabilidades procurando que la cohesión sea lo más alta posible.

Consecuencias:

- Se incrementa la claridad y facilita la comprensión, se simplifica el mantenimiento.
- Implica casi siempre bajo acoplamiento, incrementando la reutilización.

Ejemplo:

Este ejemplo es una contradicción, es decir, el ejemplo va **en contra** del patrón de Alta Cohesión.



Patrones relacionados:

Bajo Acoplamiento.

Bajo Acoplamiento.

Problema:

¿Cómo soportar baja dependencia e incrementar la reutilización? Acoplamiento indica que tan fuertemente está conectada una clase con otra, tiene conocimiento de, o influye sobre otra clase, una clase con bajo acoplamiento no depende de otras clases.

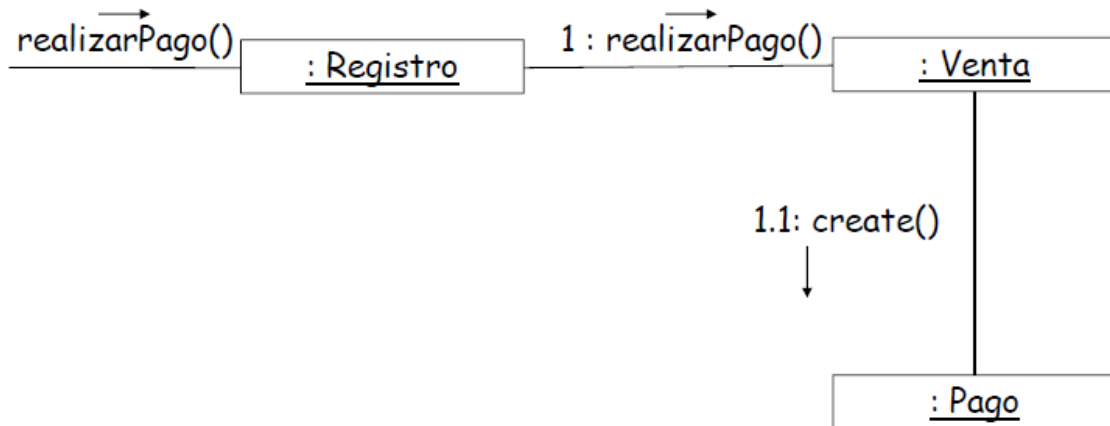
Solución:

Asignar responsabilidades de tal manera que el acoplamiento sea el menor posible.

Consecuencias:

- Mejor comprensión de las clases aisladas.
- Facilitan la reutilización de código.
- No afectan los cambios en otros componentes.

Ejemplo:



Patrones relacionados:

Variaciones Protegidas.

Controlador.

Problema:

¿Quién debe manejar eventos del sistema? Un controlador es un objeto responsable del manejo de los eventos del sistema, que no pertenece a la interfaz del usuario, el controlador recibe la solicitud del servicio desde la capa GUI y coordina su realización delegando a otros objetos.

Solución:

Responsabilidades para el manejo de mensajes de eventos del sistema a una clase que:

- Representa al conjunto del sistema o negocio (**Controlador Fachada**).
- Representa algo del mundo real que está activo (**Controlador de Roles**).
- Representa un administrador artificioso para todos los eventos del sistema (**Controlador Caso de Uso**).

Ejemplo:

En el PDV hay varias operaciones del sistema como finalizarVenta(), introducirArticulo() o introducirArticuloDevuelto().

Si siguiendo el patrón Controlador podríamos asignar la responsabilidad de manejar estos eventos a una clase que represente al sistema como **Registro (Controlador Fachada)**.

O si hay muchos eventos de sistema, también podríamos asignársela a **controladores de los distintos casos de uso** como: ProcesarVentaControlador o GestionarDevolucionesControlador.

Si el Registro asume **demasiadas responsabilidades pierde la cohesión** (el compromiso).

Consecuencias:

- Potencial para reutilizar interfaces conectables.
- Razonamiento sobre el funcionamiento de los casos de uso, es decir, valida la secuencia de operaciones, capturando información de estado.

Patrones relacionados:

Command, Fachada, Capas, Fabricación Pura.

Más Patrones GRASP.

Más patrones GRASP:

1. **Polimorfismo.**
2. **Fabricación Pura.**
3. **Indirección.**
4. **Variaciones Protegidas.**

Polimorfismo.

Problema:

¿Cómo manejar alternativas basadas en el tipo? ¿Cómo crear componentes conectables?

Solución:

Cuando el comportamiento relacionado varía según el tipo (clase) asigne la responsabilidad para el comportamiento utilizando operaciones polimórficas a los tipos para los que varía el comportamiento.

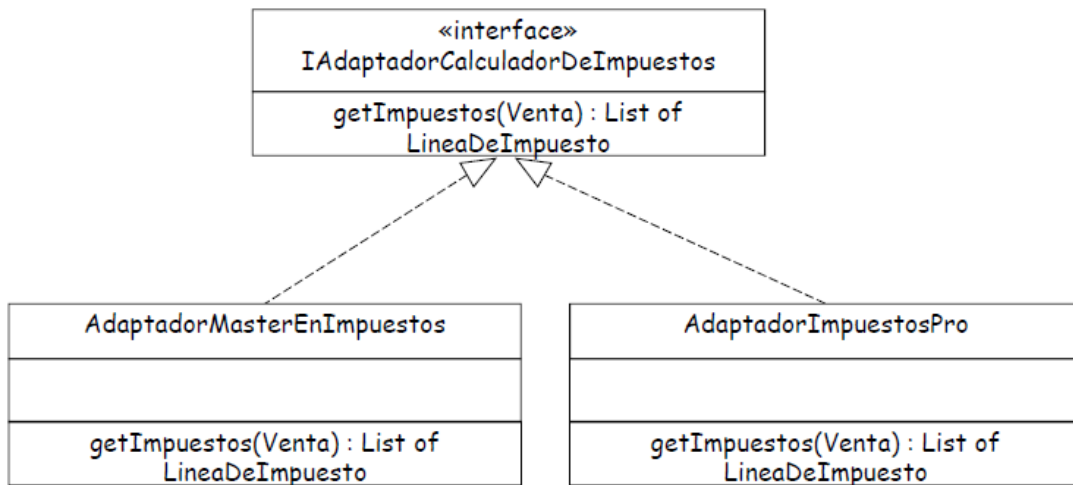
- **Corolario:** Evítense las comprobaciones acerca del tipo de un objeto y procure no utilizar la lógica condicional.

Consecuencias:

- Se añaden fácilmente extensiones necesarias para nuevas variaciones.

- Las nuevas implementaciones se introducen sin afectar a los clientes.

Ejemplo:



Patrones relacionados:

Variaciones Protegidas, varios de los patrones de diseño GoF como Adaptador, Command, Composite, Proxy, Estado y Estrategia.

Fabricación Pura.

Problema:

¿Qué objetos deberían tener la responsabilidad cuando no se quiere violar los objetivos de alta cohesión y bajo acoplamiento?

Solución:

Asigne un conjunto de responsabilidades altamente cohesivas a una clase artificial que no representa un concepto del dominio.

Consecuencias:

- Se soporta alta cohesión puesto que las responsabilidades se factorizan en una clase de grano fino.
- El potencial para reutilizar aumenta.

Ejemplo:

- Se necesita soporte para almacenar la **Venta** en una base de datos se podría suponer según el **Experto** que la propia clase debería tener la responsabilidad.
- La tarea implica un número amplio de operaciones de bases de datos no relacionadas con las ventas (**Baja cohesión**).
- Además tiene que **acoplarse** con el interfaz de la base de datos.

- Y por último la tarea de almacenar objetos puede ser necesaria para más objetos.
- Podríamos crear una clase **AlmacenamientoPersistente**.

Contraindicaciones:

Llevar el principio al extremo.

Patrones relacionados:

Bajo Acoplamiento, Alta Cohesión, los GoF (Adaptador, Command, Estrategia, etc) y prácticamente el resto de patrones de diseño.

Indirección.

Problema:

¿Dónde asignar una responsabilidad para evitar el acoplamiento directo entre dos objetos?

Solución:

Asigne la responsabilidad a un objeto intermedio que medie entre otros componentes para que no estén acoplados directamente, el intermediario crea un indirección.

Consecuencias:

- Disminuye el acoplamiento entre componentes.

Ejemplos:

- Muchos de los patrones existentes son especializaciones de Indirección.
- Está relacionado con Variaciones Protegidas, Bajo Acoplamiento, muchos GoF como Adaptador, Puente, Fachada, Observador y Mediador.

Variaciones Protegidas.

Problema:

¿Cómo diseñar componentes de manera que las variaciones en ellos no tengan repercusión en otros elementos?

Solución:

Asigne responsabilidades para crear una interfaz estable alrededor de los puntos de variaciones previstas o inestabilidad.

Consecuencias:

- Se añaden fácilmente las extensiones para nuevas variaciones.
- Se pueden introducir nuevas implementaciones sin afectar a los clientes.

- Se reduce el acoplamiento.

Ejemplo:

Mediante la Indirección de la interfaz y el polimorfismo se consigue una protección frente a las variaciones en las interfaces externas, integrando el sistema con otros sistemas externos.

Patrones relacionados:

La mayoría de los patrones son mecanismos de Variaciones Protegidas. Se relaciona con Polimorfismo, Indirección y la mayor parte de los GoF.

Patrones GoF.

Patrones publicados por Gamma, Helm, Johnson y Vlossodes en 1995: patrones de la pandilla de los cuatro (gang of four).

Esta serie de patrones permiten ampliar el lenguaje, aprender nuevos estilos de diseño y además se introducirá más notación UML.

Existen 23 patrones GoF de los que 15 se utilizan con frecuencia y 8 se verán en este tema.

Se clasifican según su propósito en patrones:

- **De creación:** Factoría Abstracta, Singleton (Creación de objetos).
- **Estructurales:** Adaptador, Composite, Fachada (Composición de clases y objetos).
- **De comportamiento:** Observador, Estrategia (Modo en que las clases y objetos interactúan y se reparten las responsabilidades).

Adaptador-Wrapper (estructural).

Problema:

¿Cómo resolver interfaces incompatibles, o proporcionar una interfaz estable para componentes parecidos con diferentes interfaces?

Solución:

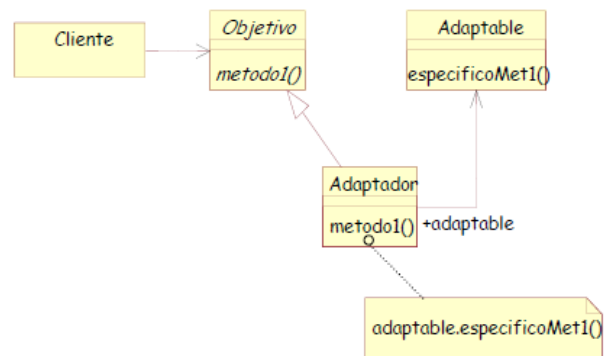
Convierta la interfaz de una clase en otra interfaz que es la que esperan los clientes, permite que cooperen clases que de otra forma no podrían cooperar por tener interfaces incompatibles.

Ejemplo:

El **Adaptador** ofrece **Variaciones Protegidas** en paquetes de terceras partes mediante un objeto Indirección que aplica **Interfaces** y **Polimorfismo**.

Participantes:

- **Objetivo.** Define la interfaz que utiliza el Cliente.
- **Cliente.** Colabora con objetos que se ajustan a la interfaz Objetivo.
- **Adaptable.** Define una interfaz existente que necesita ser adaptada.
- **Adaptador.** Adapta la interfaz de Adaptable a la interfaz Objetivo.



Consecuencias:

- Posibilidad de reutilizar las clases que han sido pensadas para este Objetivo pero que tenían interfaces no compatibles con el sistema en desarrollo.
- Un adaptador de objetos permite adaptar una clase y sus subclases, puede funcionar para varios adaptables.

Factoría (creación).

Problema:

¿Quién es el responsable de la creación de objetos cuando existen consideraciones especiales, como una lógica de creación compleja o el deseo de separar las responsabilidades de la creación para mejorar la cohesión?

Solución:

Cree un objeto **Fabricación Pura** denominado Factoría (Factoría Concreta) que resuelva la creación cuando existan consideraciones especiales.

Ejemplo:

¿Quién crea Adaptador y cuál debería crearse?

Si se encarga a algún objeto de dominio excedería la lógica de la aplicación o disminuiría su cohesión. Resulta mejor asignar a una **FactoríaDeServicios** dicha responsabilidad, así se crea el Adaptador dinámicamente. De esta manera se podría cambiar de Adaptador sin cambiar código.

Consecuencias:

- Separar la responsabilidad de la creación compleja en objetos de apoyo.
- Ocultar la lógica de creación potencialmente compleja.

- Permitir introducir estrategias para mejorar el rendimiento de la gestión de memoria.

Factoría Abstracta (creación).

Problema:

¿Cómo crear familias de clases relacionadas que implementan una interfaz común?

Solución:

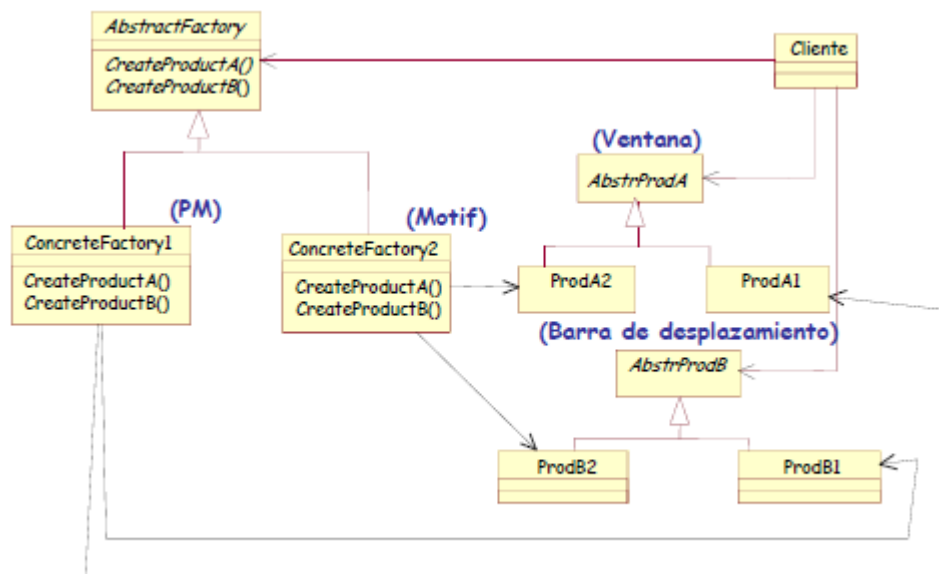
Proporcione una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

Ejemplo:

Aplicación para crear interfaces de usuario que admita múltiples estándares como Motif o Presentation Manager, la Factoría Abstracta define una interfaz para cada una de las utilidades básicas, cada subclase concreta define la utilidad apropiada para su estándar particular.

Participantes:

- **FabricaAbstracta.** Declara la interfaz para operaciones que crean objetos productos abstractos.
- **FabricaConcreta.** Implementa las operaciones para crear objetos productos concretos.
- **ProductoAbstracto.** Declara una interfaz para un tipo de objeto producto.
- **ProductoConcreto.** Define un objeto producto para que sea creado por la fábrica correspondiente, implementa ProductoAbstracto.
- **Cliente.** Usa las interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.



Aplicabilidad:

Un sistema es independiente de cómo sus productos son creados, compuestos y representados.

Un sistema debe ser configurado como una familia de productos de entre varias.

Una familia de objetos producto relacionados está diseñada para ser utilizada conjuntamente y se necesita hacer cumplir esta restricción.

Se quiere proporcionar una librería de productos y se quiere revelar sólo la interfaz no la implementación.

Consecuencias:

- Aísla los clientes de las clases concretas de implementación.
- Favorece la consistencia entre productos.
- Pero es difícil soportar nuevas clases de productos.

Singleton (creación).

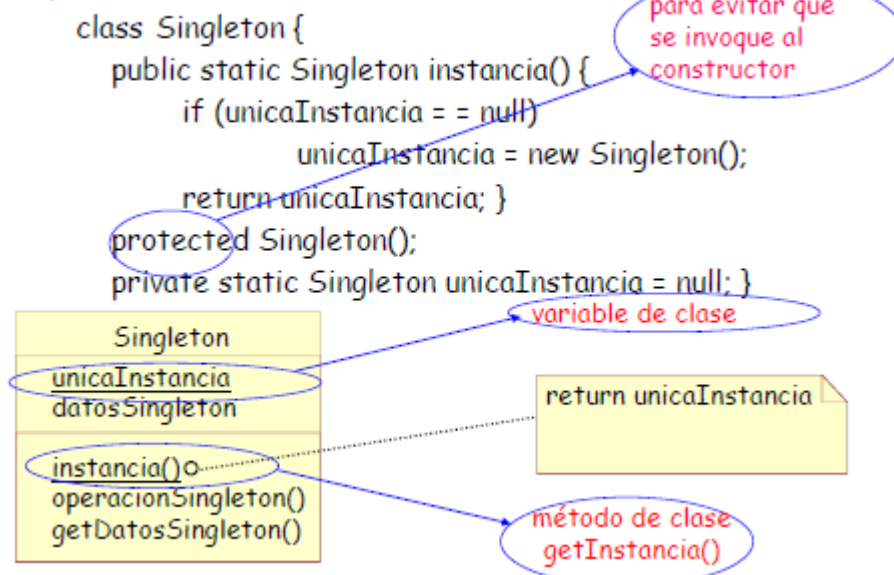
Problema:

¿Cómo podemos asegurar que una clase tiene exactamente una única instancia y que ésta sea fácilmente accesible?

Solución:

Defina un método estático de la clase que devuelva el singleton.

• Implementación:



Consecuencias:

- Permite el manejo de objetos únicos y que sean accesibles a otros objetos.
- Acceso controlado a la única instancia.

Patrones relacionados:

Se utiliza a menudo para Factoría y Fachada.

Estrategia (comportamiento).

Problema:

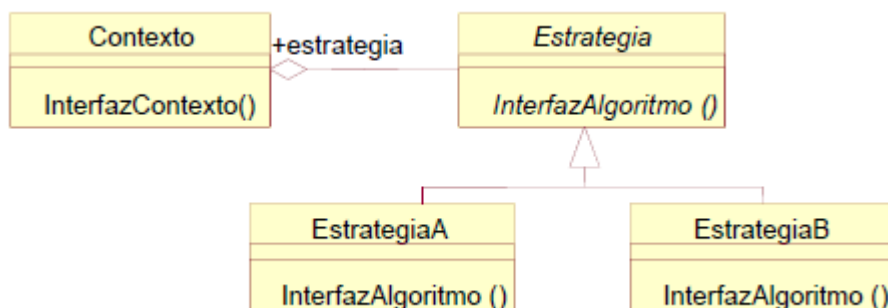
¿Cómo Diseñar diversos algoritmos o políticas que estén relacionadas? ¿Cómo diseñar que éstos puedan cambiar?

Solución:

Defina cada algoritmo, política o estrategia en una clase independiente con una interfaz común.

Participantes:

- **Estrategia**. Interfaz común a todos los algoritmos permitidos.
- **Estrategia concreta**. Implementa los algoritmos correspondientes, usando la interfaz Estrategia.
- **Contexto**. Mantiene una referencia al objeto Estrategia, se configura con un objeto Estrategia Concreta.



Consecuencias:

- Configurar una clase con uno de varios comportamientos posibles.

- Se necesitan diferentes variantes de un algoritmo para un mismo comportamiento.
- Una clase define muchos comportamientos que aparecen como sentencias CASE en sus métodos, elimina sentencias CASE.
- El cliente puede elegir entre diferentes estrategias o implementaciones pero debe conocer los detalles que las diferencian.

Patrones relacionados:

Se crea mediante una Factoría que se accede como Singleton.

Composite (estructural).

Problema:

¿Cómo Tratar un grupo o una estructura compuesta del mismo modo que un objeto atómico?

Solución:

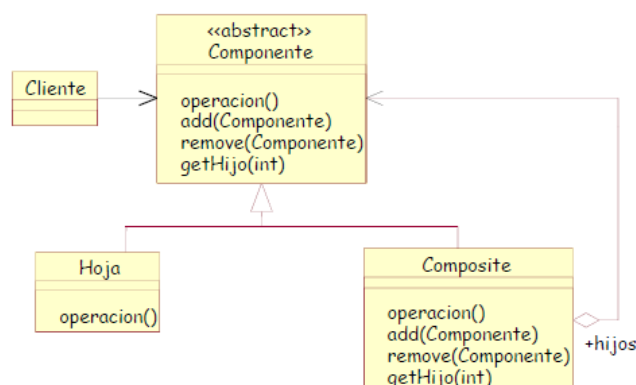
Defina las clases para los objetos compuestos y atómicos de manera que implementen el mismo interfaz.

Ejemplo:

Se puede crear una estrategia que implemente la misma interfaz y que al mismo tiempo englobe varias estrategias, se van añadiendo en distintos puntos.

Participantes:

- **Componente.** Declara la interfaz para los objetos de la composición e implementa también los métodos que determinan el comportamiento de las diferentes clases tanto compuestas como hijas y permite tener acceso a componentes en la estructura recursiva.
- **Hoja.** Define el comportamiento de los objetos que no tienen hijos en la composición.
- **Compuesto.** Maneja los hijos de los diferentes componentes compuestos y determina el comportamiento de aquellos componentes capaces de tener hijos.
- **Cliente.** Manipula los objetos de la composición a través del componente.



Consecuencias:

- Define jerarquías parte/todo.
- Los clientes ignoran la diferencia entre objetos compuestos y objetos individuales que los forman, pueden tratar objetos primitivos y compuestos de **modo uniforme**.
- Jerarquía con clases que modelan objetos primitivos y objetos compuestos, se permite composición recursiva.
- Es fácil añadir nuevos tipos de componentes.

Fachada (estructural).

Problema:

Se requiere una interfaz común para un conjunto de implementaciones o interfaces dispares.

Solución:

Defina un único punto de conexión de un subsistema, este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los clientes. (Controlador de fachada).

Consecuencias:

- Oculta a los clientes los componentes del subsistema.
- Promueve acoplamiento débil entre el subsistema y los clientes.
- No impide que las aplicaciones usen las clases del subsistema.

Observador (comportamiento).

Problema:

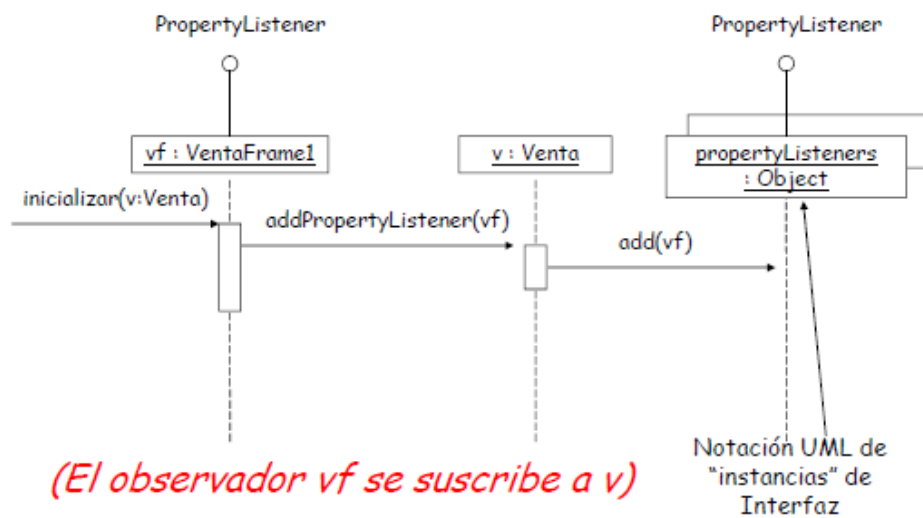
Diferentes tipos de objetos suscriptores están interesados en el cambio de estado o eventos de un emisor y quieren reaccionar cada uno a su manera, además el emisor quiere mantener **bajo acoplamiento** con los suscriptores.

Solución:

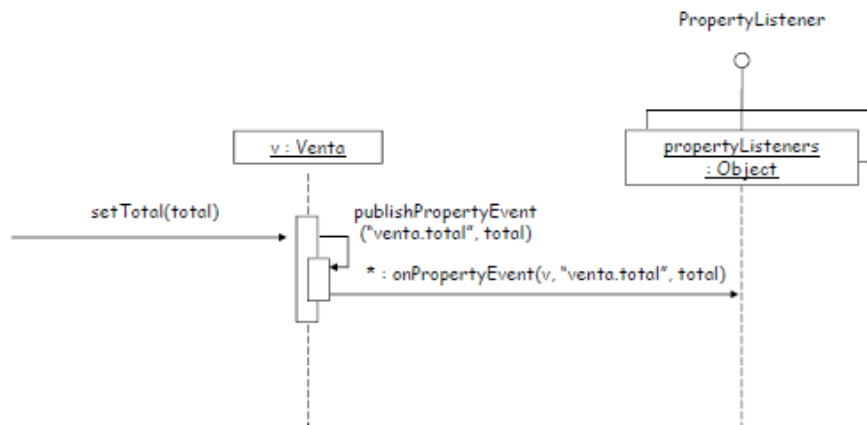
Defina una interfaz “suscriptor” u “oyente” (**listener**) para que objetos registrados como suscriptores de un emisor la implementen y se les pueda notificar cuando ocurren cambios de estado o eventos del emisor.

Pasos a seguir:

- Se define una interfaz con la operación para actualizar los objetos ante cambios en el emisor.
- Se define la ventana(s) que implementa la interfaz y el método anterior.
- Cuando se inicializa la ventana se le pasa la instancia de la clase de la que está mostrando sus datos.
- La ventana se registra o suscribe a la instancia de la clase por medio de la operación de suscripción definida en la clase.



- La clase no conoce a los objetos ventana, sólo se acopla con una interfaz.
- Como emisor, la clase, cuando cambia los datos itera sobre todos los objetos registrados y se lo notifica a cada uno.



(v publica un evento a todos los suscriptores)

Consecuencias:

- Proporciona un modo de acoplar débilmente los objetos en términos de comunicación.
- Los emisores conocen a los suscriptores sólo a través de una interfaz y desconocen incluso el número de objetos con que se comunica.
- Los suscriptores pueden registrarse o darse de baja dinámicamente.