Patrones de diseño

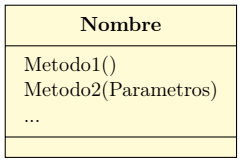
6 de noviembre de 2019

Índice general

Capítulo 1

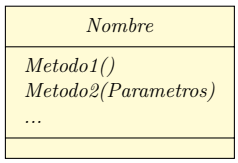
Diagramas

1.1. Módulo



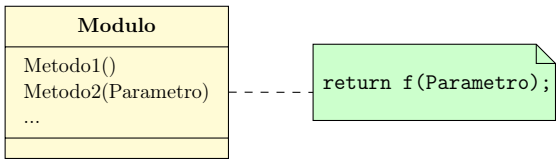
Un módulo es una unidad de implementación de software que provee una unidad coherente de funcionalidad. También se puede definir como una unidad de implementación de software que provee un conjunto de servicios. Parnas define módulo como una asignación de trabajo para un programador o un grupo de programadores.

1.2. Interfaz



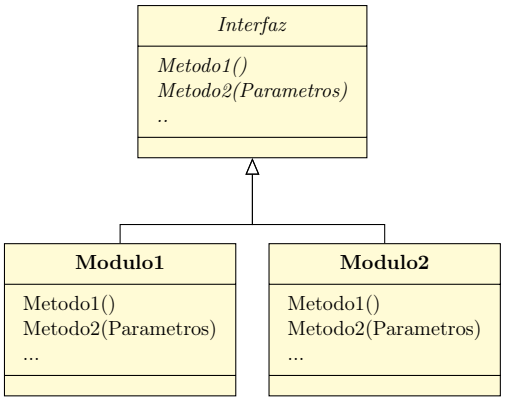
La interfaz de un módulo puede definirse como el conjunto de servicios que el módulo exporta. También puede definirse como todas las interacciones que tiene el módulo con su entorno (es decir, los otros módulos).

1.3. Implementación



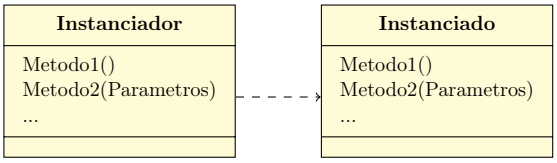
La implementación de un módulo es la forma en que se logra que la interfaz funcione según lo perciben los otros módulos.

1.4. Herencia



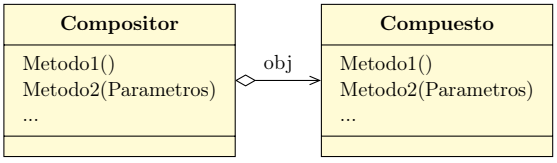
Sean A y B dos módulos, decimos que B es un heredero de A si toda subrutina de la interfaz de A es una subrutina de la interfaz de B.

1.5. Instanciacion

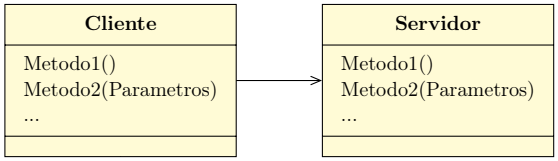


Un objeto es una instancia de un módulo.

1.6. Composición



1.7. Asociación



Capítulo 2

Patrones de creación

2.1. Abstract Factory

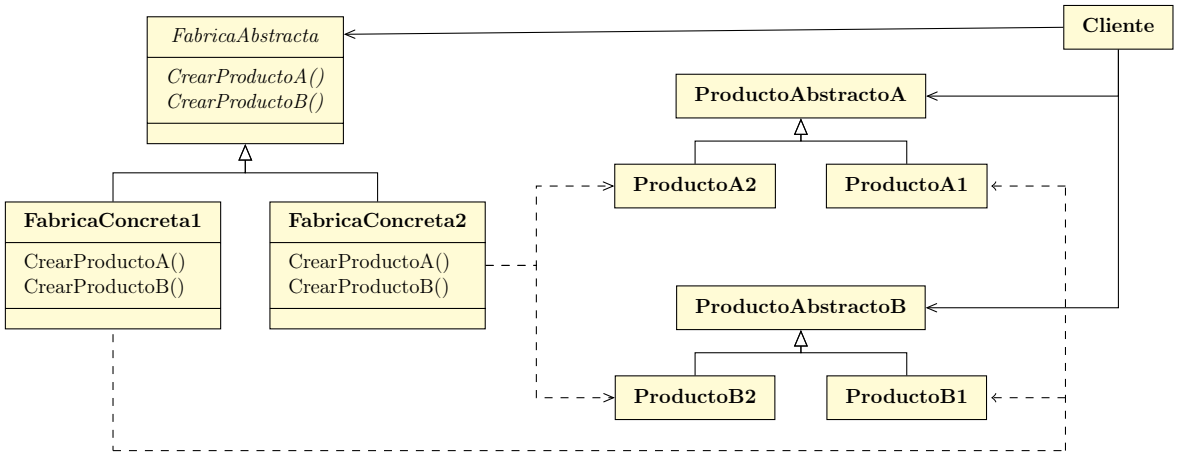
2.1.1. Propósito

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

2.1.2. Aplicabilidad

- Cuando el sistema debe ser independiente de cómo se crean, componen y representan sus productos.
- Cuando el sistema debe ser configurado con una familia de productos de entre varias.
- Cuando una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.
- Cuando quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones.

2.1.3. Estructura



2.1.4. Participantes

FabricaAbstracta

- Declara una interfaz para operaciones que crean objetos producto abstractos.

FabricaConcreta

- Implementa las operaciones para crear objetos producto 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 la interfaz ProductoAbstracto.

Cliente

- Sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.

2.1.5. Colaboraciones

- Normalmente sólo se crea una única instancia de una clase FabricaConcreta en tiempo de ejecución. Esta fábrica concreta crea objetos producto que tienen una determinada implementación. Para crear diferentes objeto producto, los clientes deben usar una fábrica concreta diferente.
- FabricaAbstracta delega la creación de objetos producto en su subclase FabricaConcreta.

2.1.6. Consecuencias

- Aísla las clases concretas. El patrón Abstract Factory ayuda a controlar las clases de objetos que crea una aplicación. Encapsula la responsabilidad y el proceso de creación de objetos producto, aísla a los clientes de las clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas. Los nombres de las clases producto quedan aisladas en la implementación de la fábrica concreta; no aparecen en el código cliente.
- Facilita el intercambio de familias de productos. La clase de una fábrica concreta sólo aparece una vez en una aplicación. Esto facilita cambiar la fábrica concreta que usa una aplicación. Como una fábrica abstracta crea una familia completa de productos, toda la familia de productos cambia de una vez.
- Promueve la consistencia entre productos. Cuando se diseñan objetos producto en una familia para trabajar juntos, es importante que una aplicación use objetos de una sola familia a la vez. FabricaAbstracta facilita que se cumpla esta restricción.
- Es difícil dar cabida a nuevos tipos de productos. Ampliar las fábricas abstractas para producir nuevos tipos de productos no es fácil.

2.1.7. Patrones relacionados

- ~~Las clases FabricaAbstracta suelen implementarse con métodos de fabricación (patron Factory Method), pero también se pueden implementar usando prototipos (patrón Prototype).~~

- ~~Una fábrica concreta suele ser un Singleton.~~

2.1.8. Documentación

Pattern	**Nombre**		
based on	Abstract Factory		
because	Fundamentación de la elección del patrón en términos de: ▪ Los cambios que este admite y los cambios probables anticipados en el diseño concreto. ▪ Las necesidades funcionales de alguna parte del sistema. ▪ Las restricciones de diseño que se deseen imponer.		
where	Cliente	**is**	
	FabricaAbstracta	***is***	*FabricaAbstracta*
	CrearProductoA()	***is***	
	CrearProductoB()	***is***	
	FabricaConcreta1	**is**	
	CrearProductoA()	**is**	
	CrearProductoB()	**is**	
	FabricaConcreta2	**is**	
	CrearProductoA()	**is**	
	CrearProductoB()	**is**	
	ProductoAbstractoA	***is***	
	ProductoA1	**is**	
	ProductoA2	**is**	
	ProductoAbstractoB	***is***	
	ProductoB1	**is**	
	ProductoB2	**is**	
comments	Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño		

Capítulo 3

Patrones estructurales

3.1. Birdge

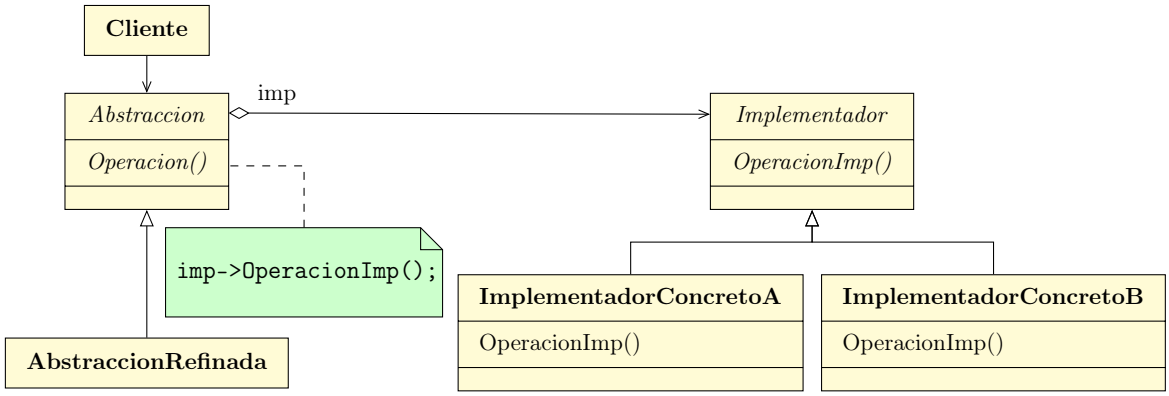
3.1.1. Propósito

Desacopla una abstracción de una implementación de manera tal que ambas puedan variar de forma independiente.

3.1.2. Aplicabilidad

- Evitar un enlace permanente entre una abstracción y su implementación. Por ejemplo, cuando debe seleccionarse o cambiarse la implementación en tiempo de ejecución.
- Evitar que los cambios en la implementación de una abstracción impacten en el código cliente; es decir, su código no tendría que ser recompilado.
- Extender de forma independiente las diferentes abstracciones y sus implementaciones.
- Compartir una implementación entre varios objetos.

3.1.3. Estructura



3.1.4. Participantes

Abstracción

- Define la interfaz de abstracción.
- Mantiene una referencia a un objeto de tipo Implementador.

Abstracción Refinada

- Extiende la interfaz definida por Abstracción.

Implementador

- Define la interfaz de las clases de implementación. Esta interfaz no tiene por que corresponderse exactamente con la de Abstracción; de hecho, ambas interfaces pueden ser muy distintas. Normalmente la interfaz Implementador solo proporciona operaciones primitivas, y Abstracción define operaciones de mas alto nivel basadas en dichas primitivas.

Implementador Concreto

- Implementa la interfaz Implementador y define su implementación concreta.

3.1.5. Colaboraciones

Abstracción redirige las peticiones del cliente a su objeto Implementador.

3.1.6. Consecuencias

- *Desacopla la interfaz y la implementación.* No une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse en tiempo de ejecución. Incluso es posible que un objeto cambie su implementación en tiempo de ejecución.

  Además, este desacoplamiento potencia una división en capas que puede dar lugar a sistemas mejor estructurados. La parte de alto nivel de un sistema sólo tiene que conocer a Abstracción y a Implementador.

- *Mejora la extensibilidad.* Podemos extraer las jerarquías de Abstracción y de Implementador de forma independiente.

- *Oculta detalles de implementación a los clientes.* Podemos aislar a los clientes de los detalles de implementación, como el compartimiento de objetos implementadores y el correspondiente mecanismo de conteo de referencias (si es que hay alguno).

3.1.7. Patrones relacionados

- El patrón Abstract Factory puede crear y configurar el Bridge.

- ~~El patrón Adapter esta orientado a conseguir que trabajen juntas clases que no están relacionadas. Normalmente se aplica a sistemas que ya han sido diseñados. El patrón Bridge, por otro lado, se usa al comenzar un diseño para permitir que abstracciones e implementaciones varíen independientemente unas de otras.~~

3.1.8. Documentación

Pattern	**Nombre en del diseño**	
based on	Bridge	
because	Fundamentación de la elección del patrón en términos de: ▪ Los cambios que este admite y los cambios probables anticipados en el diseño concreto. ▪ Las necesidades funcionales de alguna parte del sistema. ▪ Las restricciones de diseño que se deseen imponer.	
where	Cliente	**is**
	Abstraccion	***is***
	Operacion()	***is***
	AbstraccionRefinada	**is**
	Implementador	***is***
	OperacionImp()	***is***
	ImplementadorConcretoA	**is**
	OperacionImp()	**is**
	ImplementadorConcretoB	**is**
	OperacionImp()	**is**
comments	Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño	

3.2. Composite

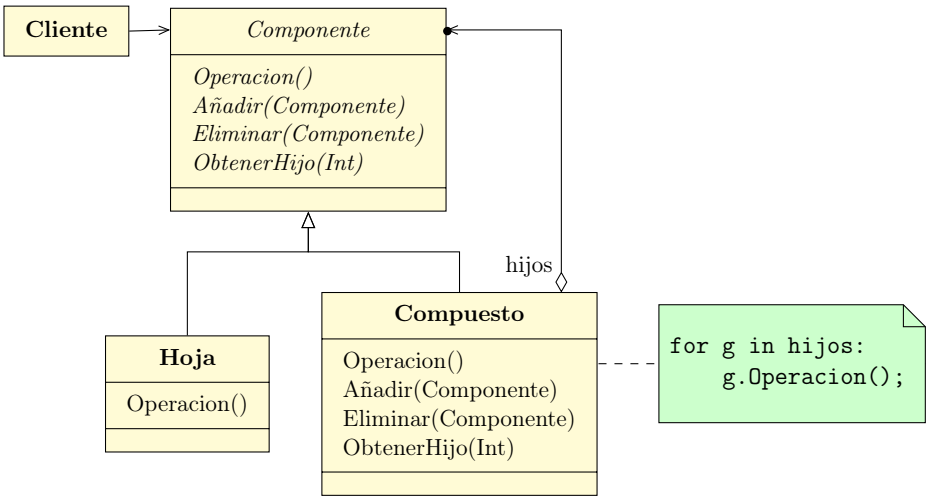
3.2.1. Propósito

Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

3.2.2. Aplicabilidad

- Representar jerarquías de objetos parte-todo.
- Abstraer las diferencias entre composiciones de objetos y objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

3.2.3. Estructura



3.2.4. Participantes

Componente

- Declara la interfaz de los objetos de la composición.
- Implementa el comportamiento predeterminado de la interfaz que es común a todas las clases.
- Declara una interfaz para acceder a sus componentes hijos y gestionarlos.
- Opcionalmente, define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.

Hoja

- Representa objetos hoja en la composición. Una hoja no tiene hijos.
- Define el comportamiento de los objetos primitivos de la composición.

Compuesto

- Define el comportamiento de los componentes que tienen hijos.
- Almacena componentes hijos.
- Implementa las operaciones de la interfaz Componente relacionadas con los hijos.

Cliente

- Manipula objetos en la composición a través de la interfaz Componente.

3.2.5. Colaboraciones

Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

3.2.6. Consecuencias

- Define jerarquías de clases formadas por objetos primitivos y compuestos. Allí donde el código espere un objeto primitivo, también podrá recibir un objeto compuesto.
- Simplifica el cliente. Los cliente pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales.

- Facilita añadir nuevos tipos de componentes. Si se definen nuevas subclases Compuesto u Hoja, éstas funcionarán automáticamente con las estructuras y el código cliente existentes. No hay que cambiar los clientes para las nuevas clases Componente.

- Puede hacer que un diseño sea demasiado general. La desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto.

3.2.7. Patrones relacionados

- ~~Muchas veces se usa el enlace al componente padre para implementar el patrón Chain of Responsibility.~~

- El patrón Decorator (Wrapper) suele usarse junto con el Composite. Cuando se usan juntos decoradores y compuestos, normalmente ambos tendrán una clase padre común. Por lo tanto, los decoradores tendrán que admitir la interfaz Componente con operaciones como Añadir, Eliminar y ObtenerHijo.

- ~~El patron Flyweight permite compartir componentes, si bien en ese caso éstos ya no pueden referirse a sus padres.~~

- Se puede usar el patrón Iterator para recorrer las estructuras definidas por el patrón Composite.

- El patrón Visitor localiza operaciones y comportamiento que de otro modo estaría distribuido en varias clases Compuesto y Hoja.

3.2.8. Documentación

<table>
<tr><td>Pattern
based on</td><td colspan="3">Nombre
Composite</td></tr>
<tr><td>because</td><td colspan="3">Fundamentación de la elección del patrón en términos de:
▪ Los cambios que este admite y los cambios probables anticipados en el diseño concreto.
▪ Las necesidades funcionales de alguna parte del sistema.
▪ Las restricciones de diseño que se deseen imponer.</td></tr>
<tr><td>where</td><td>Componente</td><td>is</td><td>Componente</td></tr>
<tr><td></td><td>Operacion()</td><td>is</td><td>Operacion()</td></tr>
<tr><td></td><td>Añadir(Componente)</td><td>is</td><td>Añadir(Componente)</td></tr>
<tr><td></td><td>Eliminar(Componente)</td><td>is</td><td>Eliminar(Componente)</td></tr>
<tr><td></td><td>ObtenerHijo(Int)</td><td>is</td><td>ObtenerHijo(Int)</td></tr>
<tr><td></td><td>Compuesto</td><td>is</td><td></td></tr>
<tr><td></td><td>Operacion()</td><td>is</td><td>Operacion()</td></tr>
<tr><td></td><td>Añadir(Componente)</td><td>is</td><td>Añadir(Componente)</td></tr>
<tr><td></td><td>Eliminar(Componente)</td><td>is</td><td>Eliminar(Componente)</td></tr>
<tr><td></td><td>ObtenerHijo(Int)</td><td>is</td><td>ObtenerHijo(Int)</td></tr>
<tr><td></td><td>Hoja</td><td>is</td><td></td></tr>
<tr><td></td><td>Operacion()</td><td>is</td><td>Operacion()</td></tr>
<tr><td>comments</td><td colspan="3">Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño</td></tr>
</table>

3.3. Wrapper/Decorator

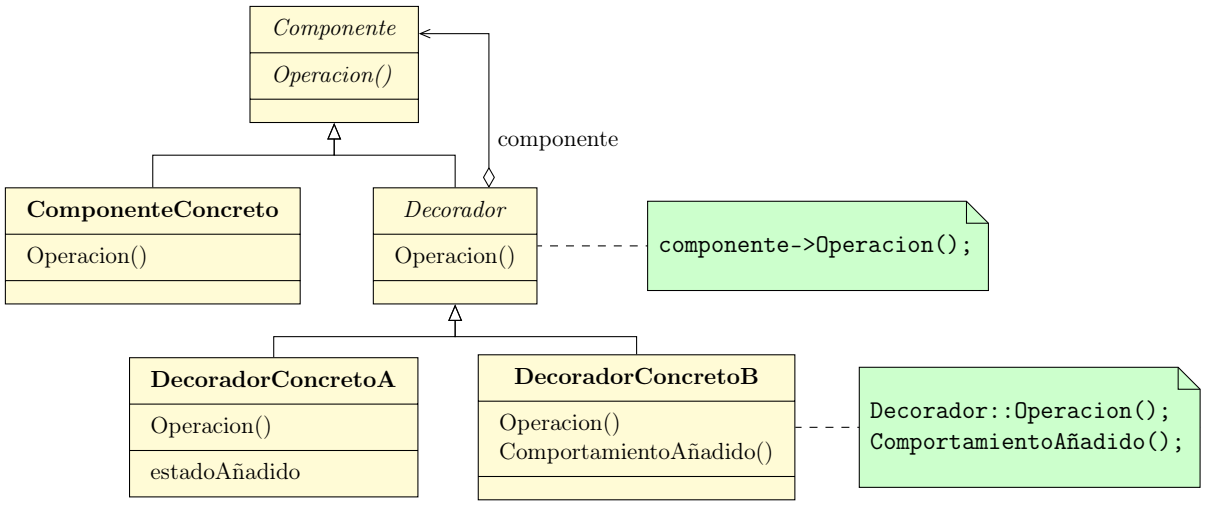
3.3.1. Propósito

Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

3.3.2. Aplicabilidad

- Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
- Para responsabilidades que pueden ser retiradas.
- Cuando la extensión mediante la herencia no es viable. A veces es posible tener un gran número de extensiones independientes, produciéndose una explosión de subclases para permitir todas las combinaciones. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada.

3.3.3. Estructura



3.3.4. Participantes

Componente

- Define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.

ComponenteConcreto

- Define un objeto al que se pueden añadir reponsabilidades adicionales.

Decorador

- Mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la interfaz Componente.

DecoradorConcreto

- Añade responsabilidades al componente.

3.3.5. Colaboraciones

El Decorador redirige peticiones a su objeto Componente. Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.

3.3.6. Consecuencias

- Mas flexibilidad que la herencia estática. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas. Por otro lado, proporcionar diferentes clases Decorador para una determinada clase Componente permite mezclar responsabilidades. Los Decoradores también facilitan añadir una propiedad dos veces.

- Evita clases cargadas de funciones en la parte de arriba de la jerarquía. En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, podemos definir primero una clase simple y añadir luego funcionalidad incrementalmente con objetos Decorador. La funcionalidad puede obtenerse componiendo partes simples. También resulta fácil definir nuevos tipos de Decoradores independientemente de las clases de objetos de las que hereden, incluso para extensiones que no hubieran sido previstas.

3.3.7. Patrones relacionados

- ~~Adapter: un decorador se diferencia de un adaptador en que el decorador sólo cambia las responsabilidades de un objeto, no su interfaz, mientras que un adaptador le da a un objeto una interfaz completamente nueva.~~

- Composite: podemos ver a un decorador como un compuesto degenerado que sólo tiene un componente. No obstante, un decorador añade responsabilidades adicionales; no está pensando para la agregación de objetos.

- Strategy: un decorador permite cambiar el exterior de un objeto; una estrategia permite cambiar sus «tripas». Son dos formas alternativas de modificar un objeto.

3.3.8. Documentación

<table>
<tr><td>Pattern</td><td colspan="3">Nombre</td></tr>
<tr><td>based on</td><td colspan="3">Wrapper</td></tr>
<tr><td>because</td><td colspan="3">Fundamentación de la elección del patrón en términos de:
Los cambios que este admite y los cambios probables anticipados en el diseño concreto.
Las necesidades funcionales de alguna parte del sistema.
Las restricciones de diseño que se deseen imponer.</td></tr>
<tr><td rowspan="11">where</td><td>Componente</td><td>is</td><td>Componente</td></tr>
<tr><td>Operacion()</td><td>is</td><td></td></tr>
<tr><td>ComponenteConcreto</td><td>is</td><td></td></tr>
<tr><td>Operacion()</td><td>is</td><td></td></tr>
<tr><td>Decorador</td><td>is</td><td></td></tr>
<tr><td>Operacion()</td><td>is</td><td></td></tr>
<tr><td>DecoradorConcretoA</td><td>is</td><td></td></tr>
<tr><td>Operacion()</td><td>is</td><td></td></tr>
<tr><td>DecoradorConcretoB</td><td>is</td><td></td></tr>
<tr><td>Operacion()</td><td>is</td><td></td></tr>
<tr><td>comments</td><td colspan="3">Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño</td></tr>
</table>

Capítulo 4

Patrones de comportamiento

4.1. Command

4.1.1. Propósito

Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones.

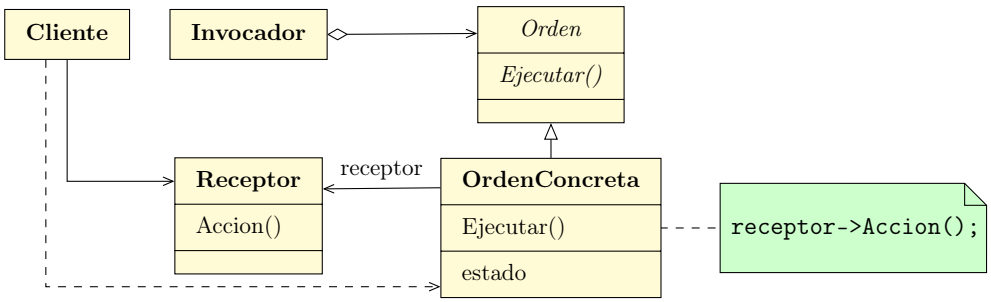
4.1.2. Aplicabilidad

- Parametrizar objetos con una acción a realizar. Los objetos Orden son un sustituto orientado a objetos para las funciones callback.
- Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo. Un objeto Orden puede tener un tiempo de vida independiente de la petición original.
- Permitir deshacer. La operación Ejecutar de Orden puede guardar en la propia orden el estado que anule sus efectos. Debe añadirse a la interfaz Orden una operación Deshacer que anule los efectos de una llamada anterior a Ejecutar. Las órdenes ejecutadas se guardan en una lista que hace las veces de historial.

- Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída del sistema. Aumentando la interfaz Orden con operaciones para cargar y guardar se puede mantener un registro persistente de los cambios. Recuperarse de una caída implica volver a cargar desde el disco las órdenes guardadas y volver a ejecutarlas con la operación Ejecutar.

- Estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas. Dicha estructura es común en los sistemas de información que permiten transacciones.

4.1.3. Estructura



4.1.4. Participantes

Orden

- Declara una interfaz para ejecutar una operación.

OrdenConcreta

- Define un enlace entre un objeto Receptor y una acción.

- Implementa Ejecutar invocando la correspondiente operación u operaciones del Receptor.

Cliente

- Crea un objeto OrdenConcreta y establece su receptor.

Invocador

- Le pide a la orden que ejecute la petición.

Receptor

- Sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como Receptor.

4.1.5. Colaboraciones

- El cliente crea un objeto OrdenConcreta y especifica su receptor.
- Un objeto Invocador almacena el objeto OrdenConcreta.
- El invocador envía una petición llamando a Ejecutar sobre la orden. Cuando las órdenes se pueden deshacer, OrdenConcreta guarda el estado para deshacer la orden antes de llamar a Ejecutar.
- El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.

4.1.6. Consecuencias

- Orden desacopla el objeto que invoca la operación de aquél que sabe cómo realizarla.
- Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto.
- Se pueden ensamblar órdenes en una orden compuesta. En general, las órdenes compuestas son una instancia del patrón Composite.
- Es mas fácil añadir nuevas órdenes, ya que no hay que cambiar las clases existentes.

4.1.7. Patrones relacionados

- Se puede usar el patrón Composite para implementar ordenes compuestas.
- ~~Un memento puede mantener el estado que necesitan las órdenes para anular (deshacer) sus efectos.~~
- ~~Una orden que debe ser copiada antes de ser guardada en el historial funciona como un Prototype.~~

4.1.8. Documentación

Pattern	**Nombre**		
based on	Command		
because	Fundamentación de la elección del patrón en términos de: ▪ Los cambios que este admite y los cambios probables anticipados en el diseño concreto. ▪ Las necesidades funcionales de alguna parte del sistema. ▪ Las restricciones de diseño que se deseen imponer.		
where	Invocador	**is**	
	Orden	***is***	*Orden*
	Ejecutar()	***is***	*Ejecutar()*
	OrdenConcreta	**is**	
	Ejecutar()	**is**	Ejecutar()
	Receptor	**is**	
	Accion()	**is**	
	estado	**is**	
comments	Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño		

4.2. Iterator

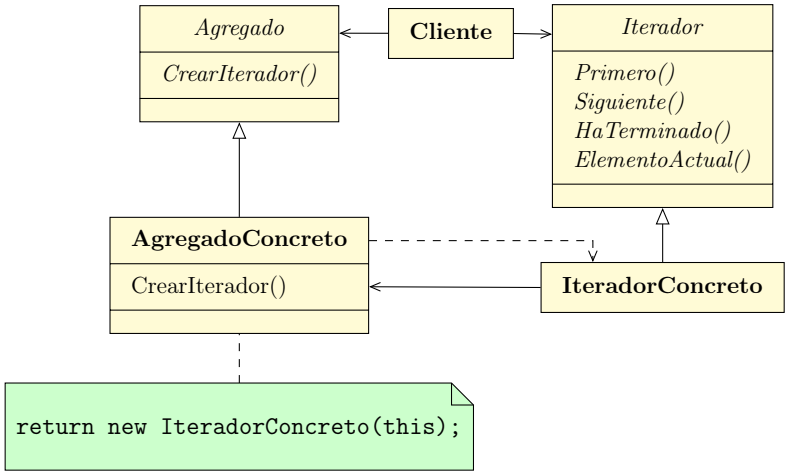
4.2.1. Propósito

Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

4.2.2. Aplicabilidad

- Acceder al contenido de un objeto agregado sin exponer su representación interna.
- Permitir varios recorridos sobre objetos agregados.
- Proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas.

4.2.3. Estructura



4.2.4. Participantes

Iterador

- Define una interfaz para recorrer los elementos y acceder a ellos.

IteradorConcreto

- Implementa la interfaz Iterador.
- Mantiene la posición actual en el recorrido del agregado.

Agregado

- Define una interfaz para crear un objeto Iterador.

AgregadoConcreto

- Implementa la interfaz de creación de Iterador para devolver una instancia del IteradorConcreto apropiado.

4.2.5. Colaboraciones

- Un IteradorConcreto sabe cuál es el objeto actual del agregado y puede calcular el objeto siguiente en el recorrido.

4.2.6. Consecuencias

- Permite variaciones en el recorrido de un agregado. Los agregados pueden recorrerse de muchas formas. Los iteradores facilitan cambiar el algoritmo de recorrido: basta con sustituir la instancia de iterador por otra diferente. También se pueden definir subclases de Iterador para permitir nuevos recorridos.
- Los iteradores simplifican la interfaz de los agregados. La interfaz de recorrido de Iterador elimina la necesidad de una interfaz parecida en Agregado, simplificando así la interfaz del agregado.
- Se puede hacer más de un recorrido a la vez sobre un agregado. Un iterador mantiene su propio estado del recorrido, por lo tanto, es posible estar realizando más de un recorrido al mismo tiempo.

4.2.7. Patrones relacionados

- Composite: los iteradores suelen aplicarse a estructuras recursivas como los compuestos.
- ~~Factory Method: los iteradores polimórficos se basan en métodos de fabricación para crear instancias de las subclases apropiadas de Iterator.~~
- ~~El patrón Memento suele usarse conjuntamente con el patrón Iterator. Un iterador puede usar un memento para representar el estado de una interacción. El iterador almacena el memento internamente.~~

4.2.8. Documentación

Pattern			**Nombre**
based on			Iterator
because			
where	Cliente	**is**	
	Agregado	***is***	*Agregado*
	crearIterador()	***is***	*crearIterador()*
	AgregadoConcreto	**is**	
	crearIterador()	**is**	crearIterador()
	Iterador	***is***	*Iterador*
	primero()	***is***	*primero()*
	siguente()	***is***	*siguente()*
	haTerminado()	***is***	*haTerminado()*
	elementoActual()	***is***	*elementoActual()*
	IteradorConcreto	**is**	
	primero()	**is**	primero()
	siguente()	**is**	siguente()
	haTerminado()	**is**	haTerminado()
	elementoActual()	**is**	elementoActual()
comments	Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño		

4.3. Strategy

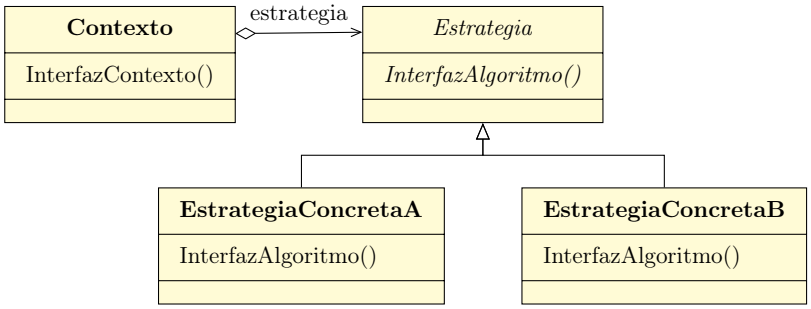
4.3.1. Propósito

Defina 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.

4.3.2. Aplicabilidad

- Cuando muchas clases relacionadas difieren sólo en su comportamiento, las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- Cuando se necesitan distintas variantes de un algoritmo.
- Cuando un algoritmo usa datos que los clientes no deberían conocer, puede usarse el patrón Strategy para evitar exponer estructuras de datos complejas.
- Cuando una clase define muchos comportamientos, en vez de tener muchos condicionales podemos mover las ramas de éstos a su propia clase Estrategia.

4.3.3. Estructura



4.3.4. Participantes

Estrategia

- Declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta.

EstrategiaConcreta

- Implementa el algoritmo usando la interfaz de Estrategia.

Contexto

- Se configura con un objeto EstrategiaConcreta.
- Mantiene una referencia a un objeto Estrategia.
- Puede definir una interfaz que permita a la Estrategia acceder a sus datos.

4.3.5. Colaboraciones

- Estrategia y Contexto interactúan para implementar el algoritmo elegido. Un contexto puede pasar a la estrategia todos los datos requeridos por el algoritmo cada vez que se llama a éste. Otra alternativa es que el contexto se pase a sí mismo como argumento de las operaciones de Estrategia. Esto permite a la estrategia hacer llamadas al contexto cuando sea necesario.
- Un contexto redirige peticiones de los clientes a su estrategia. Los clientes normalmente crean un objeto EstrategiaConcreta, el cual pasan al contexto; por lo tanto, los clientes interactúan exclusivamente con el contexto. Suele haber una familia de clases EstrategiaConcreta a elegir por el cliente.

4.3.6. Consecuencias

- Familias de algoritmos. Las jerarquías de clases Estrategia definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. La herencia puede ayudar a sacar factor común de la funcionalidad de estos algoritmos.

- Una alternativa a la herencia. Heredar de una clase Contexto para proporcionar diferentes comportamientos liga el comportamiento al Contexto mezclando la implementación del algoritmo con el Contexto y ademas impide modificar el algoritmo dinamicamente. Encapsular el algoritmo en clases Estrategia separadas nos permite variar el algoritmo independientemente de su contexto, haciéndolo mas fácil de cambiar, comprender y extender.

- Las estrategias eliminan las sentencias condicionales. El patrón Strategy ofrece una alternativa a las sentencias condicionales para seleccionar el comportamiento deseado. Un código que contiene muchas sentencias condicionales suele indicar la necesidad de aplicar el patrón Estrategia.

- Elección de implementaciones. Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento. El cliente puede elegir entre estrategias con diferentes soluciones de compromiso tiempo y espacio.

- Los clientes deben conocer las diferentes Estrategias. El patrón tiene el inconveniente potencial de que un cliente debe comprender como difieren la Estrategias antes de seleccionar la adecuada. El patrón Strategy debería usarse sólo cuando la variación de comportamiento sea relevante a los clientes.

- Costes de comunicación entre Estrategia y Contexto. La interfaz de Estrategia es compartida por todas las clases EstrategiaConcreta, por lo tanto, es probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben a través de dicha interfaz. Esto significa que habrá veces en las que el contexto crea e inicializa parámetros que nunca se usan.

- Mayor número de objetos. Las estrategias aumentan el número de objetos de una aplicación. A veces se puede reducir este coste implementando las estrategias como objetos sin estado que puedan ser compartidos por el contexto. El patrón Flyweight describe este enfoque en más detalle.

4.3.7. Patrones relacionados

- ~~Flyweight: los objetos Estrategia suelen ser buenos pesos ligeros.~~

4.3.8. Documentación

Pattern	**Nombre**	
based on	Strategy	
because	Fundamentación de la elección del patrón en términos de: ▪ Los cambios que este admite y los cambios probables anticipados en el diseño concreto. ▪ Las necesidades funcionales de alguna parte del sistema. ▪ Las restricciones de diseño que se deseen imponer.	
where	Contexto	**is**
	InterfazContexto()	**is**
	Estrategia	***is***
	InterfazAlgoritmo()	***is***
	EstrategiaConcretaA	**is**
	InterfazAlgoritmo()	**is**
	EstrategiaConcretaB	**is**
	InterfazAlgoritmo()	**is**
comments	Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño	

4.4. Visitor

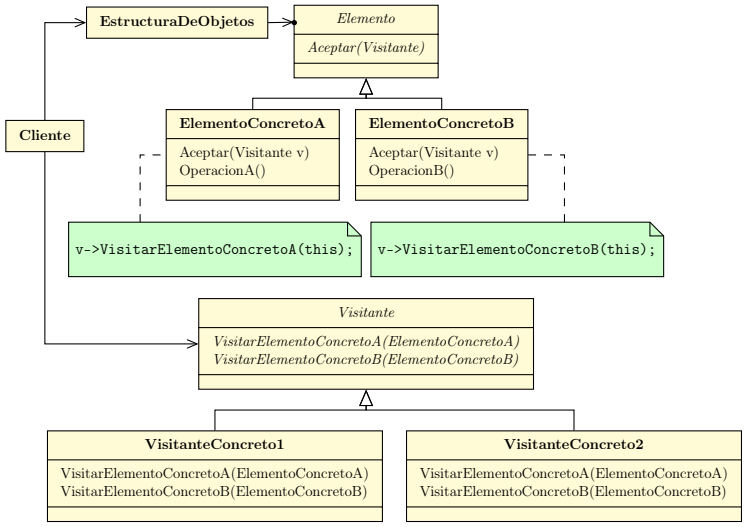
4.4.1. Propósito

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

4.4.2. Aplicabilidad

- Cuando una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta.
- Cuando se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y queremos evitar «contaminar» sus clases con dichas operaciones.
- Cuando las clases que definen la estructura de objeto rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura.

4.4.3. Estructura



4.4.4. Participantes

Visitante

- Declara una operación Visitar para cada clase de operación ElementoConcreto de la estructura de objetos. El nombre y signatura de la operación identifican a la clase que envía la petición Visitar al visitante. A continuación el visitante puede acceder al elemento directamente a través de su interfaz particular.

VisitanteConcreto

- Implementa cada operación declarada por Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura. VisitanteConcreto proporciona el contexto para el algoritmo y guarda su estado local. Muchas veces este estado acumula resultados durante el recorrido de la estructura.

Elemento

- Define una operación Aceptar que toma un visitante como argumento.

ElementoConcreo

- Implementa una operación Aceptar que toma un visitante como argumento.

EstructuraDeObjetos

- Puede enumerar sus elementos.
- Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.
- Puede ser un compuesto (Composite) o una colección, como una lista o un conjunto.

4.4.5. Colaboraciones

- Un cliente que usa el patrón Visitor debe crear un objeto VisitanteConcreto y a continuación recorrer la estructura, visitando cada objeto con el visitante.

- Cada vez que se visita a un elemento, este llama a la operación del Visitante que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.

4.4.6. Consecuencias

- Los visitantes facilitan añadir nuevas operaciones que dependen de los componentes de objetos complejos. Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante.

- Un visitante agrupa operaciones relacionadas y separa las que no lo están. El comportamiento similar no está desperdigado por las clases que definen la estructura de objetos; está localizado en un visitante. Las partes de comportamiento no relacionadas se dividen en sus propias subclases del visitante.

- Es difícil añadir nuevas clases de ElementoConcreto. Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esto representa una excepción mas que una regla.

- Visitar varias jerarquías de clases. Un Iterador puede visitar a los objetos de una estructura llamando a sus operaciones a medida que los recorre. Pero un Iterador no puede trabajar en varias estructuras de objetos con distintos tipos de elementos. El patrón Visitor no tiene esta restricción. Puede visitar objetos que no tienen una clase padre en común.

- Acumular el estado. Los visitante pueden acumular estado a medida que van visitando cada elemento de la estructura de objetos. Sin un visitante, este estado se pasaría como argumentos extra a las operaciones que realizan el recorrido.

- Romper la encapsulación. El enfoque del patrón Visitor asume que la interfaz de ElementoConcreto es lo bastante porente como para que los visitantes hagan su trabajo. Como resultado, el patrón suele obligarnos a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulación.

4.4.7. Patrones relacionados

- Composite: los visitantes pueden usarse para aplicar una operación sobre una estructura de objetos definida por el patrón Composite.

- ~~Interpreter: se puede aplicar el patrón Visitor para llevar a cabo la interpretación.~~

4.4.8. Documentación

Pattern	**Nombre**		
based on	Visitor		
because			
where	Cliente	**is**	
	EstructuraDeObjetos	**is**	
	Elemento	***is***	*Elemento*
	Aceptar(Visitante)	***is***	
	ElementoConcretoA	**is**	
	Aceptar(Visitante)	**is**	
	OperacionA()	**is**	
	ElementoConcretoB	**is**	
	Aceptar(Visitante)	**is**	
	OperacionB()	**is**	
	Visitante	***is***	
	VisitanteConcretoElementoA(ElementoConcretoA)	***is***	
	VisitanteConcretoElementoB(ElementoConcretoB)	***is***	
	VisitanteConcreto1	**is**	
	VisitarElementoConcretoA(ElementoConcretoA)	**is**	
	VisitarElementoConcretoB(ElementoConcretoB)	**is**	
	VisitanteConcreto2	**is**	
	VisitarElementoConcretoA(ElementoConcretoA)	**is**	
	VisitarElementoConcretoB(ElementoConcretoB)	**is**	
comments	Explicación coloquial de la relación entre los elementos del patrón y los elementos del diseño concreto; otros comentarios adicionales que ayuden a entender cómo se aplica el patrón de diseño		