Resumen de Objetos

Resumen para devorar el sábado en el parcial

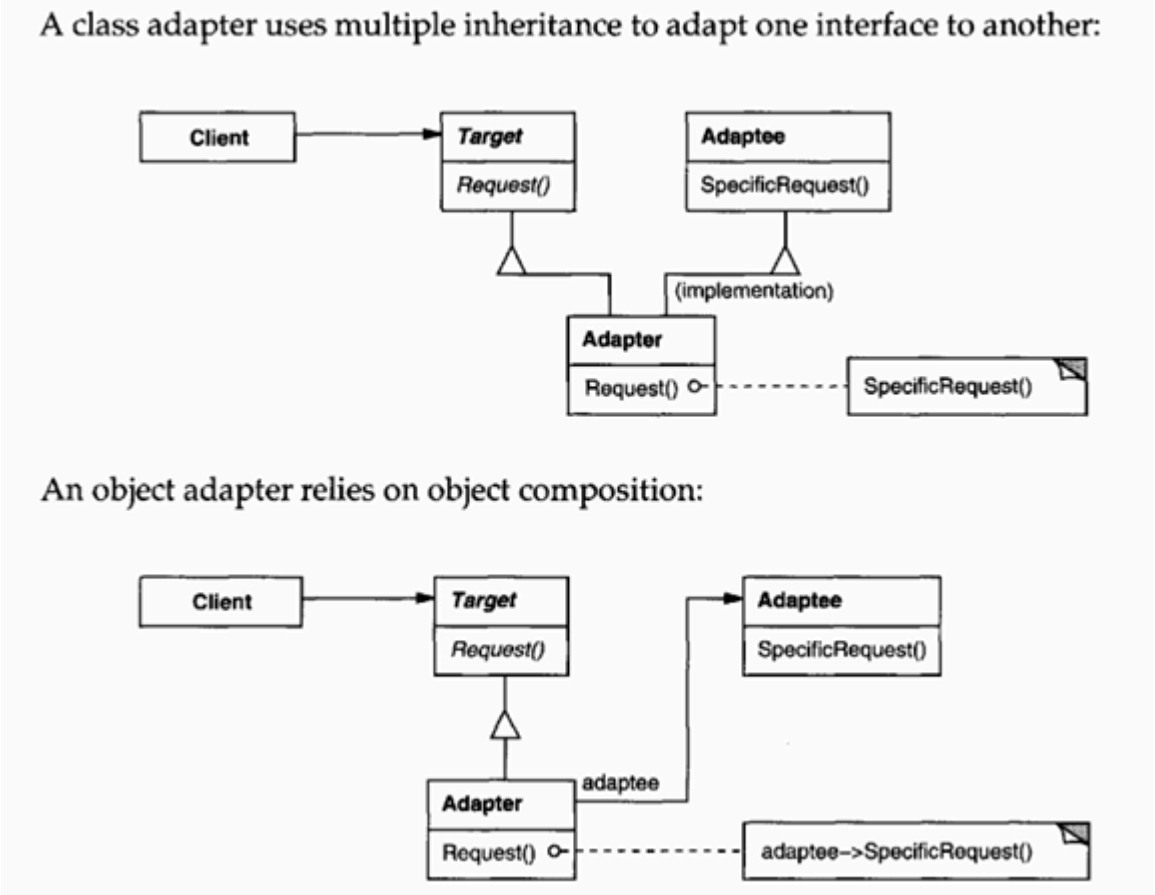
Patrones Estructurales

Adapter (Wrapper)

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 hacerlo (ya que poseen interfaces incompatibles).

Estructura



Cuándo debería ser usado

- Se quiere usar una clase existente y su interfaz no concuerda con la que se necesita.
- Se quiere crear una clase reutilizable que coopere con clases no tienen porqué tener interfaces compatibles.

Consecuencias de usarlo

- Un Adaptador de Clases no nos servirá cuando lo que queremos es adaptar una clase y todas sus subclases.
- Permite que el Adaptador redefina comportamiento del Adaptado.

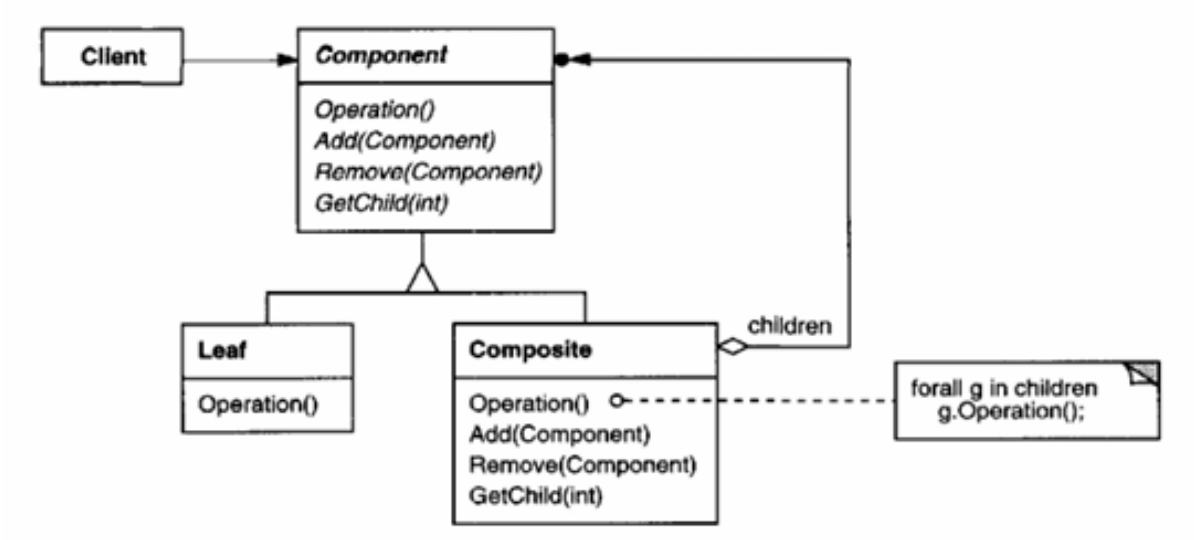
- Se introduce un solo objeto Adaptador y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.

Composite

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.

Estructura



Cuándo debería ser usado

- Se quiere representar jerarquías de objetos parte-todo.
- Se quiere hacer que los Clientes no diferencien entre un objeto compuesto y uno individual. De esa forma, el Cliente puede tratar a cualquier objeto de la estructura de manera uniforme.

Consecuencias de usarlo

- El patrón define jerarquías de clases formadas por objetos primitivos y compuestos. Los primitivos pueden componerse para generar uno compuesto que sea más complejo, esto hace que si el código en alguna parte espera un primitivo, también pueda recibir uno compuesto.
- Simplifica el código del cliente al no tener que diferenciar si está tratando con un objeto primitivo o uno compuesto.
- Facilita añadir nuevos tipos de objetos ya sean compuestos o primitivos ya que se adaptan automáticamente con la estructura del patrón y el código que conlleva el mismo.
- Puede generar un diseño muy general.

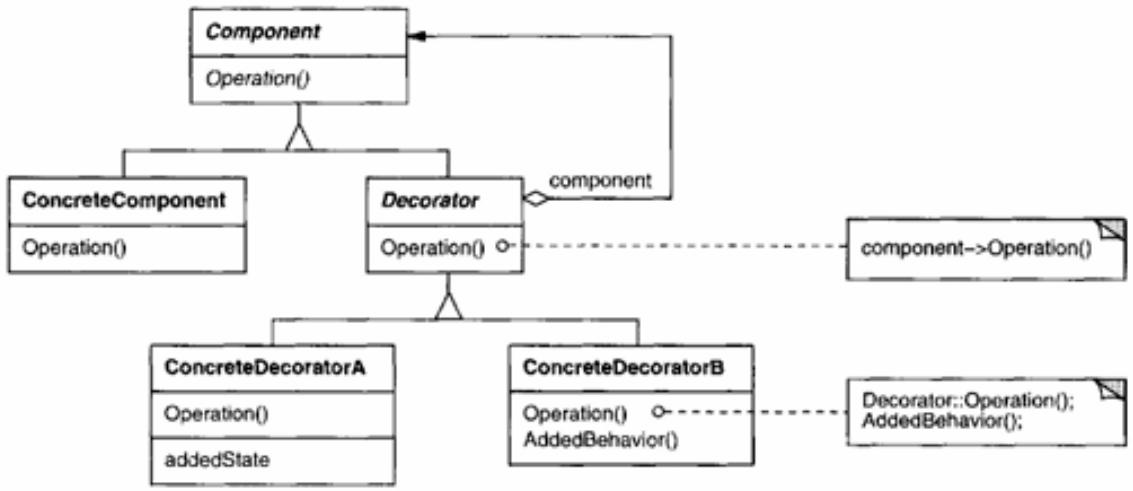
- Hace difícil restringir los componentes de un compuesto, es decir, quizás se quiere que un compuesto tenga ciertos primitivos y esto es difícil de controlar. Por eso, con este patrón no se puede confiar en el sistema de tipos, para esos chequeos, estos se deben de realizar por nosotros en tiempo de ejecución.

Decorator (Wrapper)

Propósito

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

Estructura



Cuándo debería ser usado

- Se quiere añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
- Existen responsabilidades que pueden ser retiradas y añadidas dinámicamente.
- La extensión mediante la herencia no es viable. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada.

Consecuencias de usarlo

- Da más flexibilidad que la herencia estática. Esto se ve a la hora de añadir y eliminar responsabilidades.
- Evita la generación de clases cargadas de funciones en la parte superior de la jerarquía ya que este patrón posee un enfoque de pagar sólo por aquello que se necesita.
- No existe una identidad de Objetos, es decir, un Decorador y su Componente no son idénticos. Un Decorador se comporta como un envoltorio transparente para el Componente.

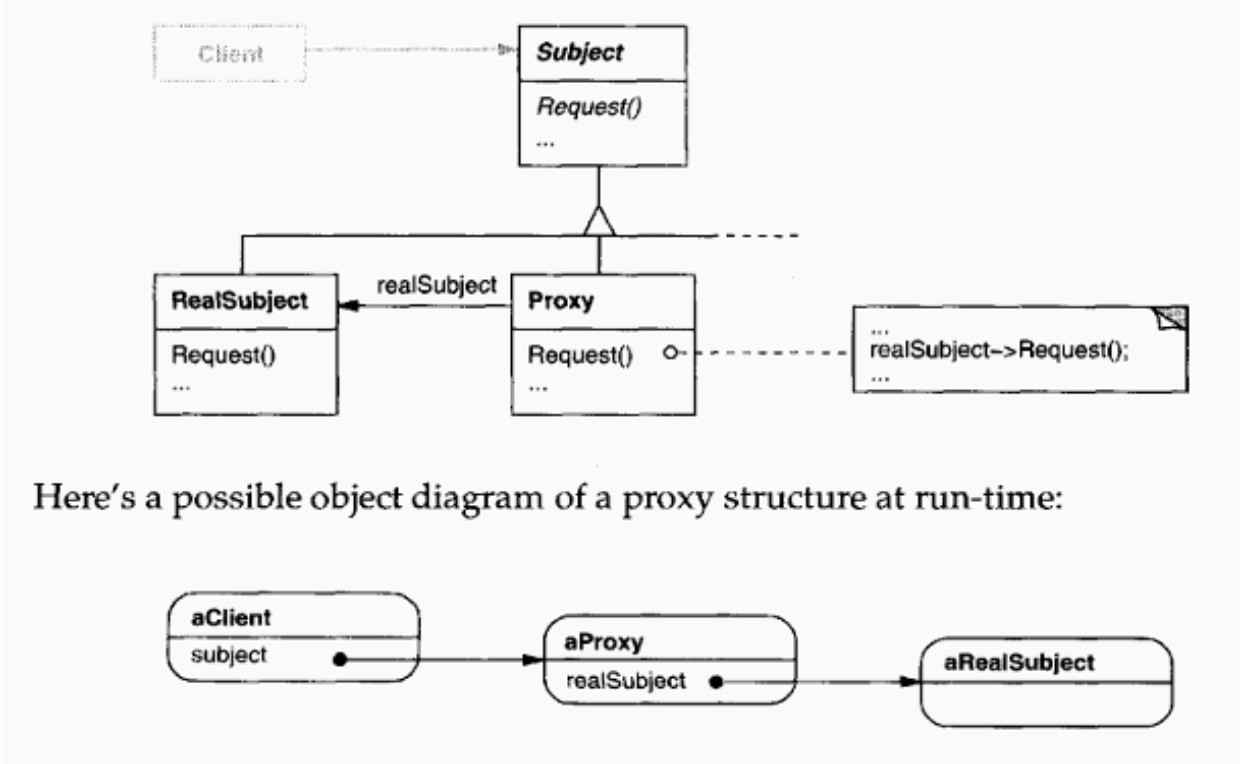
- Un diseño que use este patrón normalmente va a ser un diseño formado por muchos objetos pequeños parecidos. Esto hace que los diseños sean más difíciles de entender y depurar.

Proxy (Surrogate)

Propósito

- Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.

Estructura



Cuándo debería usarlo

- **Proxy Remoto**
  - Proporciona un representante local de un objeto situado en otro espacio de direcciones.
- **Proxy Virtual**
  - Crea objetos costosos por encargo.
- **Proxy de Protección**
  - Controla el acceso al objeto original. Estos proxys son útiles cuando los objetos deberían tener distintos permisos de acceso.
- **Referencia Inteligente**
  - Sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto.

Consecuencias de usarlo

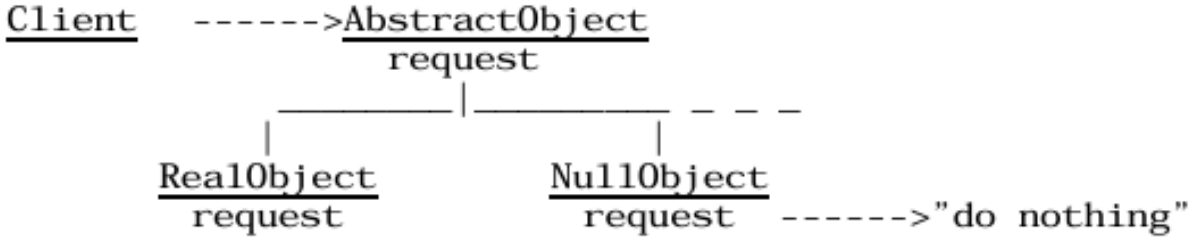
- El patrón introduce un nivel de indirección al acceder a un objeto. Esta indirección adicional tiene diferentes usos según el tipo de Proxy que se use:
    - **Proxy Remoto**
        - Puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente.
    - **Proxy Virtual**
        - Puede llevar a cabo optimizaciones tales como crear un objeto por encargo.
    - **Proxys de Protección y Referencias Inteligentes**
        - Permiten realizar tareas de mantenimiento.
- Permite ocultar al Cliente una optimización que se denomina copia-de-escritura, y que está relacionada con la creación por encargo. Copiar un objeto grande y complejo puede ser muy costoso. Si ese objeto no se modifica, no hace falta hacer ese gasto de copia. El Proxy permite posponer ese proceso de copia para hacerlo únicamente cuando sea modificado.

Null Object (Stub)

Propósito

- Proporciona un sustituto para otro objeto que comparta la misma interfaz pero que no haga nada. El objeto nulo encapsula las decisiones de implementación de cómo "no hacer nada" y oculta esos detalles de sus colaboradores.

Estructura



Cuándo debería usarlo

- Un objeto requiere un colaborador. El patrón de Objeto Nulo no introduce esta colaboración, sino que hace uso de una colaboración que ya existe.
- Algunas instancias de colaboradores deben hacer nada.
- Desea que los clientes puedan ignorar la diferencia entre un colaborador que proporciona un comportamiento real y uno que no hace nada.
- Desea poder reutilizar el comportamiento de no hacer nada para que varios clientes que necesiten este comportamiento funcionen de manera consistente.

Consecuencias de usarlo

- Define jerarquías de clases que consisten en objetos reales y objetos nulos.

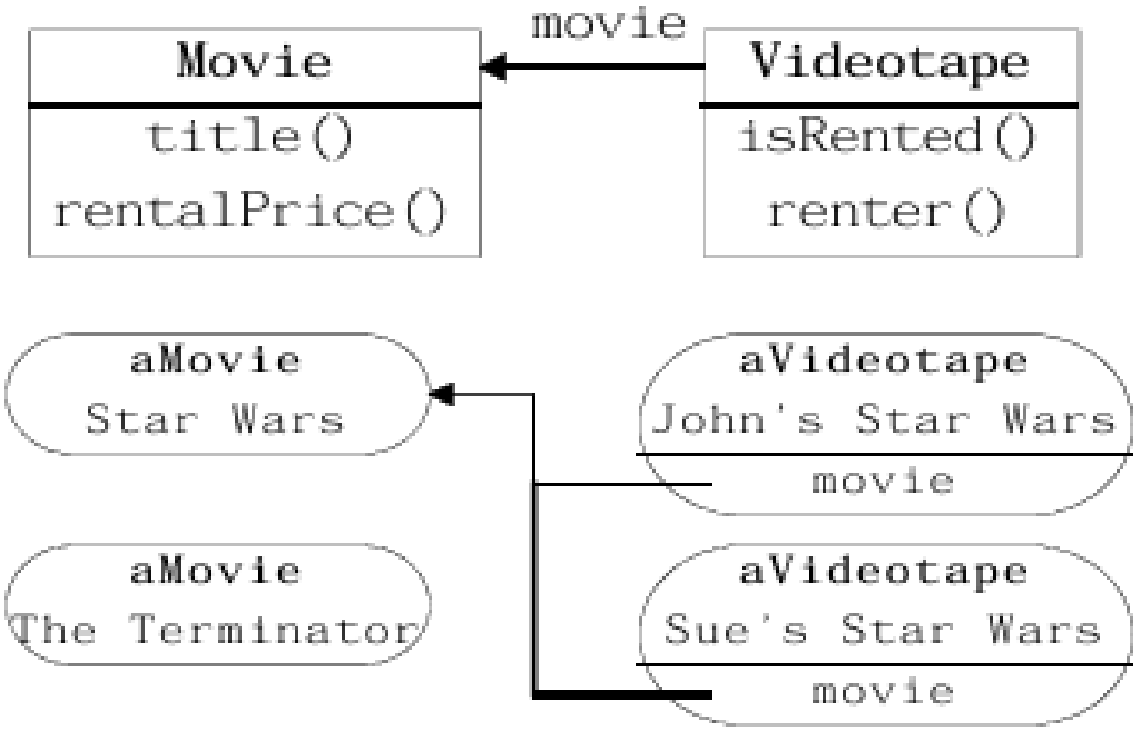
- Hace que el código del cliente sea simple. Los clientes pueden tratar a los colaboradores reales y a los colaboradores nulos de manera uniforme.
- Encapsula el código de no hacer nada en el objeto nulo. El código de no hacer nada es fácil de encontrar.
- Hace que el código de no hacer nada en el objeto nulo sea fácil de reutilizar.
- Hace que el comportamiento de no hacer nada sea difícil de distribuir o mezclar en el comportamiento real de varios objetos colaboradores.
- Siempre actúa como un objeto de no hacer nada. El Objeto Nulo no se transforma en un Objeto Real.

Type Object (Power Type, Item Descriptor, Metaobject)

Propósito

- Desacoplar instancias de sus clases para que esas clases puedan implementarse como instancias de una clase.
- Permite que se creen nuevas "clases" dinámicamente en tiempo de ejecución.
- Permite que un sistema proporcione sus propias reglas de verificación de tipos y puede llevar a sistemas más simples y pequeños.

Estructura



Cuándo debería usarlo

- Las instancias de una clase necesitan agruparse según sus atributos y/o comportamientos comunes.

- La clase necesita una subclase para cada grupo a fin de implementar los atributos/comportamientos comunes de ese grupo.
- La clase requiere una gran cantidad de subclases y/o la variedad total de subclases que pueda ser necesaria sea desconocida.
- Desee poder crear nuevos agrupamientos en tiempo de ejecución que no fueron previstos durante el diseño.
- Desee poder cambiar la subclase de un objeto después de haber sido instanciado sin tener que mutarlo a una nueva clase.
- Desee poder anidar agrupamientos recursivamente, de modo que un grupo sea en sí mismo un elemento de otro grupo.

Consecuencias de usarlo

- Creación de clases en tiempo de ejecución: El patrón permite la creación de nuevas "clases" en tiempo de ejecución. Estas nuevas clases no son realmente clases, sino instancias llamadas TypeObjects.
- Evita la explosión de subclases: El sistema ya no necesita numerosas subclases para representar diferentes tipos de objetos.
- Oculta la separación de instancia y tipo: Los clientes de un objeto no necesitan ser conscientes de la separación entre el objeto y su TypeObject.
- Cambio dinámico de tipo: El patrón permite que el objeto cambie dinámicamente su TypeObject, lo que tiene el efecto de cambiar su clase. Esto es más simple que mutar un objeto a una nueva clase.
- Subclasificación independiente: TypeClass y Class pueden ser subclasificadas independientemente.
- Múltiples TypeObjects: El patrón permite que un objeto tenga múltiples TypeObjects donde cada uno define una parte del tipo del objeto. El objeto debe entonces decidir qué comportamiento de tipo delegar a qué TypeObject.
- Complejidad del diseño: El patrón divide un objeto lógico en dos clases. Su relación, una cosa y su tipo, es difícil de entender.
- Complejidad de la implementación: El patrón mueve las diferencias de implementación fuera de las subclases y dentro del estado de las instancias de TypeObject.
- Gestión de referencias: Cada objeto debe mantener una referencia a su TypeObject. Así como un objeto sabe cuál es su clase, un objeto sabe cuál es su TypeObject.

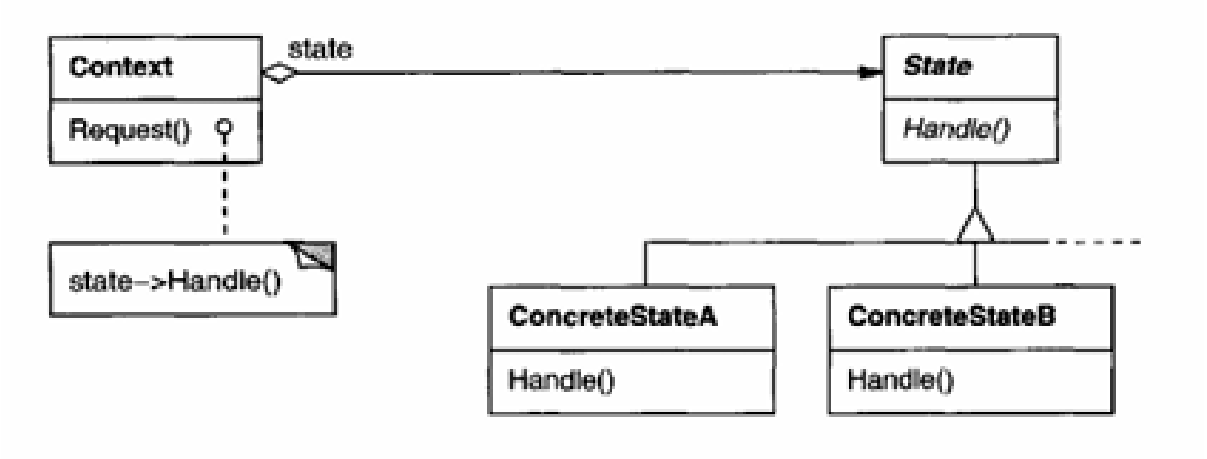
Patrones de Comportamiento

State (Objects for States)

Propósito

- Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Estructura



Cuándo debería usarlo

- El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Muchas veces se ven representados los estados del objeto en esas estructuras condicionales (Descubrimiento de clases).

Consecuencias de usarlo

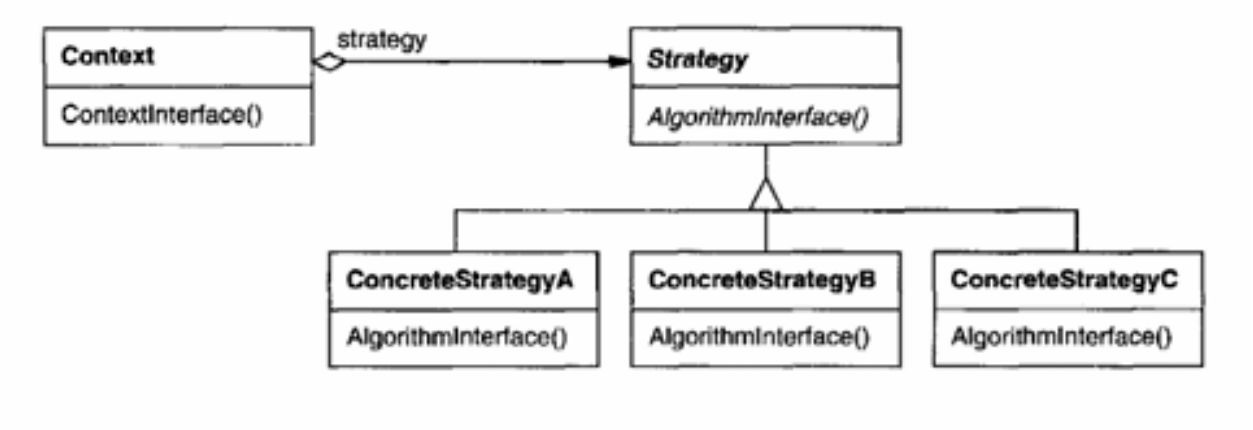
- Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados. Esto hace que incremente el número de clases y que el diseño sea menos compacto pero, esta distribución que ofrece el patrón es realmente buena para evitar grandes condicionales que hacen a nuestro código menos legible.
- Hace explícitas las transiciones entre estados que protegen al Contexto de estados internos inconsistentes.
- Los objetos Estado pueden compartirse, generando que no tengan estado intrínseco, sino que solo comportamiento.

Strategy (Policy)

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.

Estructura



Cuándo debería usarlo

- Muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- Se necesitan distintas variantes de un algoritmo.
- Un algoritmo usa datos que los clientes no deberían conocer. Se usa para evitar exponer estructuras de datos complejas y dependientes del algoritmo.
- Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones. Evitamos los condicionales representando las ramas como clases Estrategia.

Consecuencias de usarlo

- Las jerarquías de Estrategias definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. Esta herencia puede ayudar a sacar factor común de la funcionalidad de los algoritmos.
- Es una alternativa a la herencia que permite encapsular el algoritmo en clases Estrategias separadas de modo que podamos variar el algoritmo independientemente de su contexto, facilitando el cambio, la comprensión y la extensión.
- Las Estrategias eliminan las sentencias condicionales en cuánto al comportamiento que representan.
- Brindan al Cliente la capacidad de poder elegir entre Estrategias con diferentes soluciones para un comportamiento específico.
- Como inconveniente se ve necesario que el Cliente conozca las Estrategias y cómo difieren las mismas para poder seleccionar la más adecuada. Esto genera que los Clientes puedan estar expuestos a cuestiones de implementación.
- Se genera un coste de comunicación entre Estrategia y Contexto, esto se debe a que la interfaz de la Estrategia es una sola para todas las concretas, por lo tanto, es probable que algunas Estrategias Concretas no hagan uso de toda la información que reciben por parámetro de parte del Contexto.

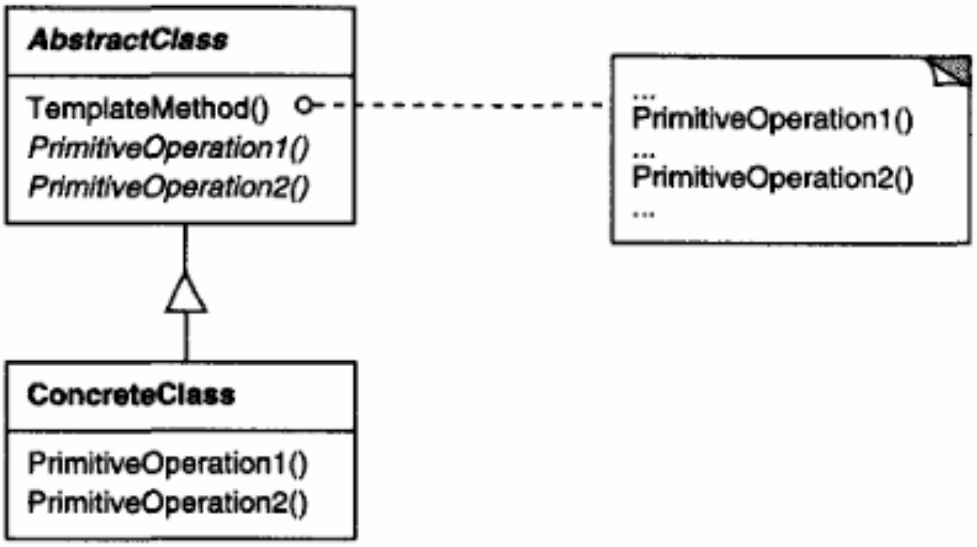
- Aumenta el número de Objetos dentro de una aplicación. Se puede solucionar haciendo que los objetos actúen como estrategias compartidas.

Template Method

Propósito

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

Estructura



Cuándo debería usarlo

- Se quiera implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar.
- Cuando el comportamiento repetido de varias subclases debería factorizar y ser localizado en una clase común para evitar código duplicado.
- Para controlar las extensiones de las subclases.

Consecuencias de usarlo

- Generan reutilización del código y factorización del mismo.

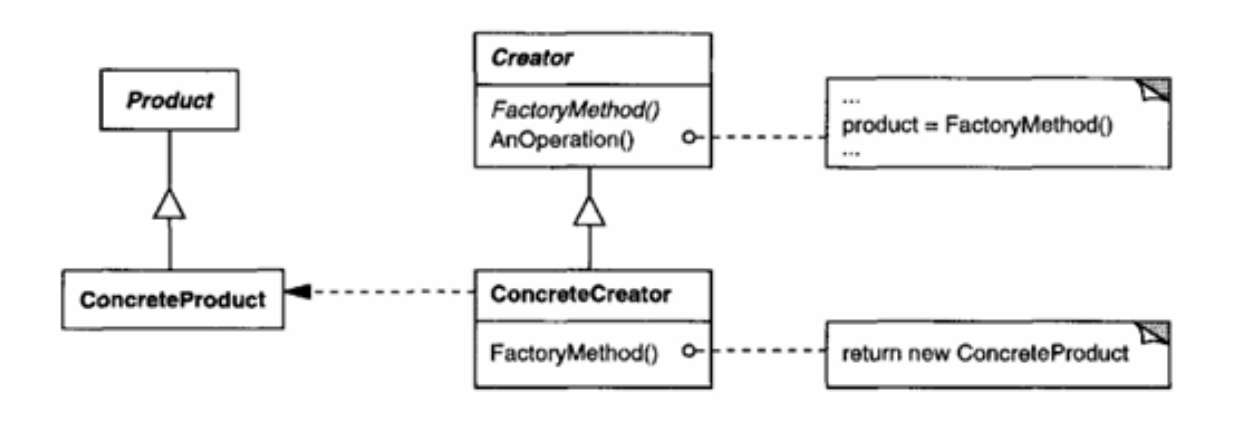
Patrones Creacionales

Factory Method (Virtual Constructor)

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.

Estructura



Cuándo debería usarlo

- 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 qué subclase de auxiliar concreta es en la que se delega.

Consecuencias de usarlo

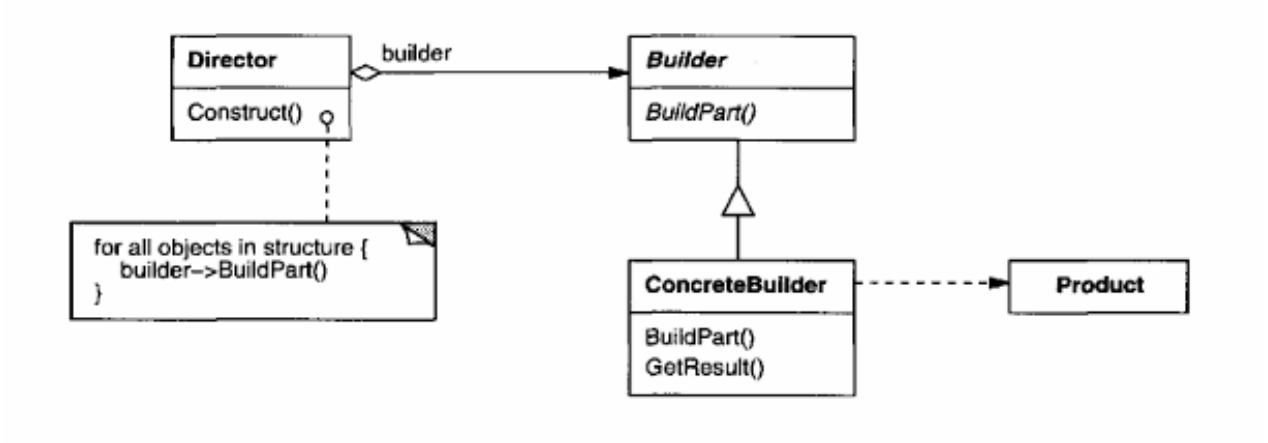
- Proporciona enganches para las subclases. Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente.
- Conecta jerarquías de clases paralelas. Esto ocurre cuando una clase delega alguna de sus responsabilidades a una clase separada. Hay veces que el cliente puede encontrar útiles los métodos de construcción.

Builder

Propósito

- Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Estructura



Cuándo debería usarlo

- El algoritmo para crear un objeto complejo debería ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.
- La creación de los objetos tiene una secuencia de pasos definida.

Consecuencias de usarlo

- Permite variar la representación interna de un producto. El Constructor proporciona al Director una interfaz abstracta para construir el Producto. Esta interfaz permite que el Constructor oculte la representación y la estructura interna del Producto, a su vez, también oculta el modo en que el Producto está ensamblado.
- Aísla el código de construcción y representación. El patrón aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos.
- Proporciona un control más fino sobre el proceso de construcción al hacerlo paso a paso y bajo el control del Director.

Frameworks

Librería de Clases

- Resuelven problemas comunes a la mayoría de las aplicaciones.
- Cada clase en la librería resuelve un problema concreto, es independiente del contexto de uso, no espera nada de nuestro código y generalmente independiente de otras clases en la librería.
- Nuestro código controla a los objetos de la librería.

Framework

- Aplicación semi completa, reusable, que puede ser especializada para producir aplicaciones a medida.

- También se puede ver como un conjunto de clases concretas y abstractas, relacionadas para proveer una arquitectura reusable para una familia de aplicaciones relacionadas.
- Los desarrolladores incorporan el framework en sus aplicaciones y lo especializan para cubrir sus necesidades.
- El Framework define el esqueleto, y el desarrollador define sus propias características para completar el esqueleto.

Frameworks de Infraestructura

- Estos frameworks ofrecen una infraestructura portable y eficiente sobre la cual construir una gran variedad de aplicaciones.
- Algunos de los focos de este tipo de frameworks son: interfaces de usuario (desktop, web, móviles), seguridad, contenedores de aplicación, procesamiento de imágenes, procesamiento de lenguaje, comunicaciones.
- Atacan problemas generales del desarrollo de software, que por lo general no son percibidos completamente por el usuario de la aplicación (es decir, resuelven problemas de los programadores, no de los usuarios).

Frameworks de Integración

- Estos frameworks se utilizan comúnmente para integrar componentes de aplicación distribuidos (p.e., la base de datos con la aplicación y ésta con su cliente liviano).
- Los frameworks de integración (o frameworks middleware) están diseñados para aumentar la capacidad de los desarrolladores de modularizar, reusar y extender su infraestructura de software para que trabaje transparentemente en un ambiente distribuido.
- El mercado de los frameworks de integración es muy activo y, al igual que ciertos frameworks de infraestructura, se han vuelto commodities (mercancía de uso común).

Frameworks de Aplicación

- Estos frameworks atacan dominios de aplicación amplios que son pilares fundamentales de las actividades de las empresas.
- Ejemplos de frameworks enterprise son aquellos en el dominio de los ERP (Enterprise Resource Planning), CRM (Customer Relationship Management) Gestión de documentos, Cálculos financieros.
- Los problemas que atacan derivan directamente de las necesidades de los usuarios de las aplicaciones y por tanto hacen que el retorno de la inversión en su desarrollo/adquisición sea más evidente y justificado.
- Un framework enterprise puede encapsular el conocimiento y experiencia de muchos años de una empresa, transformándose en la clave de su ventaja competitiva y su más preciado capital.

Frozenspots

- Aspectos del Framework que no se pueden cambiar, si estos se ven modificados será porque se aplicó Hacking.
- Se ven en los requerimientos.

Hotspots

- Puntos de extensión que nos permiten introducir variantes y así construir aplicaciones diferentes.
- Se ven en los requerimientos.

Inversión de Control

- La inversión de control se da cuando el código del framework utiliza el código que nosotros escribimos, por lo tanto ahora el control lo posee el Framework en vez de mi programa.

Plantillas y Ganchos

- Hablamos de plantillas y ganchos como una forma de separar lo que es común de lo que varía.
- El patrón template method implementa plantillas y ganchos con herencia (y mensajes a self).
- También podemos implementar plantillas y ganchos con composición (y delegación).
- En un framework, las plantillas suelen estar bajo control del desarrollador del framework (y las usa para implementar el frozenspot) que delega en el usuario la implementación/instanciación de los ganchos.

Hook Methods

- Herramienta que nos permite definir Hotspots. No es un Hotspot.

Template Method

- Herramienta que nos permite definir Frozenspots. No es un Frozenspot.