Ejercicios 6 7 8

Indica las características principales de los patrones de diseño más conocidos , indica para que valen y un caso de uso para cada uno de ellos

Patrones de diseño

- Patrones creacionales

  → Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente

- Patrones estructurales

  → Explican cómo ensablar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura

- Patrones de comportamiento

  → Tratan con algoritmos y la asignación de responsabilidades entre objetos

Patrones Creacionales

Factory Method

Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán

- Aplicabilidad:
  - Se utilizara cuando no conozcamos de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar nuestro codigo
  - Cuando queramos ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos
  - Cuando quieras ahorrar recursos del sistema mediante la reutilizacion de objetos existentes en lugar de reconstruirlos cada vez

Abstract Factory

Nos permite producir familias de objetos relacionados sin especificar sus clase concretas

- Aplicabilidad
    - Se usará cuando nuestro código deba funcionar con varias familias de productos relacionadas, pero no deseamos que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad

Builder

Nos permite construir objetos complejos paso a paso.

- Aplicabilidad
    - Cuando el constructor tenga muchos parametros
    - Cuando queramos que el código sea capaz de crear distintas representaciones de ciertos objetos que solo várian en los detalles
    - Para construir árboles con el patrón Composite u otros objetos complejos

Prototype

Nos permite copiar objetos existentes sin que el código dependa de sus clases

- Aplicabilidad
    - Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesitas copiar
    - Cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases parapoder crear objetos con una configuración especifica.

Singleton

Nos permite asegurarnos de que una clase tenga un única instancia, a la vez que proporciona un punto de acceso global a dicha instancia

- Aplicabilidad

    - Cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes;
    - Cuando necesites un control más estricto de las variables globales

Patrones estructurales

Adapter

Permite la colaboración entre objetos con interfaces incompatibles

- Aplicabilidad:
    - Cuando queramos usar una clase existente, pero cuya interfaz no sea compatible con el resto del código
    - Cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase

Bridge

Nos permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas(abstracción e implementación) que pueden desarrollarse independientemente la una de la otra

- Aplicabilidad:
    - Cuando queramos dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad
    - Cuando necesitemos extender una clase en varias dimensiones independientes
    - Cuando necesitemos poder cambiar implementaciones durante el tiempo de ejecuación

Composite

Nos permite componer objetos en estructuras de árbol y trabajar con esas estructuras con si fueran objetos individuales

- Aplicabilidad:

    - Cuando tengamos que implementar una estructura de objetos en forma de arbol
    - Cuando quieras que el código trate elementos simple y complejos de la misma forma

Decorador

Nos permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades

- Aplicabilidad
    - Cuando necesitemos asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos
    - Cuando resulte extraño o sea posible extender el comportamiento de un objeto utilizando la herencia

Facade

Proporciona una intefaz simplificada a una biblioteca, o framework o cualquier otro grupo complejo de clases

- Aplicabilidad
    - Cuando necesitamos una interfaz limitada pero directa a un subsistema complejo
    - Cuando queramos estructurar un subsistema en capas

Flyweight (peso mosca, peso ligero, cache)

Nos permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto

- Aplicabilidad
    - Solamente cuando tu programa deba soportar una enorme cantidad de objetos. que apenas quepan en la RAM disponible

Proxy

Nos permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndonos hacer algo antes o después de que la solicitud llegue al objeto original

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

Patrones de comportamiento

Chain of responsibilty

Nos permite pasar solicitudes a lo largo de una cadena de manejadores. Al Recibir una solicitud cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena

- Aplicabilidad
    - Cuando el program deba procesar distintos tipos de solicitudes de varias maneras, per los tipos exactos de solicitudes y sus secuencias no se conzcan de antemano

- Cuando se afundamental ejecutar varios manejadores en un orden especifico
- Cuando el grupo de manejadores y su orden deba cambiar durante el tiempo de ejecución

Command

Nos convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

- Aplicabilidad
  - Cuando quieras parametrizar objetos con operaciones
  - Cuando quieras poner operaciones en cola, programar su ejecución , o ejecutarlas de forma remota
  - Cuando quieras implementar operaciones reversibles

Iterator

Nos permite recorrer elementos de una colección sin exponer su representación subyacente(lista, pila, arbol, etc)

- Aplicabilidad
  - Cuando tu colección tenga una estructura de datos compleja a nivel interno, pero quieras ocultar su complejidad a los clientas(ya sea por conveniencia o por razones de seguridad)
  - Par reducir la duplicación en el código de recorrido a lo largo de tu aplicación
  - Cuando quieras que tu código pueda recorrer distintas estructuras de datos, o cuando los tipos de estas estructuras no se conozcan de antemano

Mediator

Nos permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador

- Aplicabilidad
    - Cuando resulta difçícil cambiar algunas de las clases porque están fuertemente acopladas a un puñado de otras clases
    - Cuando no puedas reutilizar un componente en un programa deferente porque sea demasiado dependiente de otros
    - Cuando te encuentres creando cientos de subclasses de componentes sólo para reutlizar un comportamiento básico en varios contextos

Memento

Nos permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación

- Aplicabilidad
    - Cuando quieras producir instántaneas del estado del objeto para poder restaurar un estado previo del objeto
    - Cuando el acceso directo a los campos, consultores o modificadores del objeto viole su encapsulación

Observer

Nos permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando

- Aplicabilida
    - Cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente
    - Cuando algunos objetos de tu aplicación deban observar a otros, pero solo durante un tiempo limitado o en casos especifícos

State

Nos permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiará su clase

- Aplicabilidad

- Cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia
- Cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase
- Cuando tengas mucho código duplicado por estados similares y transiciones de una maquina de estados basada en condiciones

Strategy

Nos permite definir una familia de algoritmos,colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables

- Aplicabilidad
  - Cuando quieras utiliza distintas variantes de un algoritmo dentro de un objeto y poder cambiar de una algoritmo a otro durante el tiempo de ejecución
  - Cuando tengas muchas clases similares que sólo se diferencian en la forma en que ejecutan cierto comportammiento
  - Para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica
  - Cuando tu clase tenga un enorme operador condicional que cambien entre distintas variantes del mismo algoritmo

Template Method

Nos define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos de algoritmo sin cambiar su estructura

- Aplicabilidad
  - Cuando quieras permitir a tus clientes que extiendan unicamente pasos particulares de una algoritmo, pero no todo el algoritmo o su estructura
  - Cuando tengas muchas clases que contengan algoritmos casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que

modificar todas las calses cuando el algoritmo cambie

Visitor

Nos permite separar algoritmos de los objetos sobre los que operan

- Aplicabilidad
    - Cuando necesite realizar una operación sobre todos los elementos de una compleja estructura de objetos

Bibliografia

1. https://refactoring.guru/es/design-patterns/