Patrón de diseño:

* Describe una solución reutilizable a un problema común en un contexto dado.
* Se enfoca en la solución y no en el problema.
* Identifica responsabilidades e interacciones entre los participantes.
* Una forma de masticar consejos sobre un tema.
* Es una solución probada y efectiva.
* No es un paradigma de programación.
* No es una bala de plata (asumir que nuestra solución técnica favorita resolverá un problema mayor).
* No resuelve un problema específico.
* No depende del lenguaje de programación implementado.

Para que sirven:

* Tener vocabulario y entendimiento común para el diseño de SW.
* Tener alternativas de diseño para que sea flexible y reutilizable.
* Construir arquitecturas de SW complejas y heterogéneas.
* Favorecen la vida y mantenibilidad de una aplicación.
* Incrementa la experiencia profesional.

Síntomas de un mal diseño:

* Rigidez. Difícil de realizar modificaciones
* Fragilidad. Un cambio impacta a otras partes de nuestro código.
* Inmovilidad. Construimos una pieza y no la podemos reutilizar.
* Viscosidad. Dificultad de que fluya la información entre los diferentes componentes.

Principios SOLID

* Principio de responsabilidad única.

Una clase debe tener bien definido su comportamiento específico.

* Principio de abierto cerrado.

Las clases deben ser abiertas para ser extensibles pero cerradas para que no impacte el comportamiento de la clase.

* Principio de sustitución de Liskov.

Las clases heredadas deben ser sustituibles por las clases padre.

* Principio de segregación de interfaces.

Crear interfaces detalladas específicas del cliente.

* Principio de inversión de dependencia.

Se debe depender de las abstracciones y no de las clases base.

Categorías de patrones (POSA):

* Patrones de arquitectura (descomposición a módulos)
  + Layers, MVC, EDA, etc.
* Patrones de diseño (como cada módulo esta implementado)
  + Factory merhod, Facade, Strategy, Observer, etc.
* Idioms/Modismos
  + Manejo de memoria, uso del lenguaje, convenciones de nombramiento, etc.

Patrones arquitectónicos:

* Leyers
  + Problema:

Diseñar un sistema cuya característica sea una mezcla de problemas de alto nivel y de bajo nivel.

El flujo de comunicación consta de peticiones desde el nivel superior al inferior y las respuestas en sentido inverso.

Si la capa inferior cambia es muy probable que tenga que cambiar las capas superiores.

* + Contexto:

Una aplicación grande que requiere descomposición.

* + Solución:

Estructurar aplicaciones que puedan ser descompuestas e grupos de subtareas, en las que cada grupo de subtareas está a un nivel particular de abstracción.

* + Ejemplo:

Arquitectura de 3 capas.

**PATRONES DE DISEÑO:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Propósito | | |
|  |  | Creación | Estructura | Comportamiento |
| Alcance | Clase | **Factory method** | **Adapter** | Interpreter  Template method |
| Objeto | **Abstract factory**  **Builder**  Prototype  **Singleton** | Adapter  Bridge  Composite  **Facade**  **Decorator**  Flyweight  Proxy | Chain of responsibility  **Command**  Iterator  Mediator  Memento  Observer  State  Strategy  Visitor |

* **Patrones de creación** (como se crea un objeto)
  + Builder
    - Problema:

Mientras más compleja la aplicación, son más complejas las clases y los objetos.

* + - Contexto:

Tenemos una aplicación que necesita un mecanismo para construir los objetos complejos y que a su vez sea independiente del objeto que lo constituye.

* + - Solución:

Definir una **instancia** para crear un objeto y dejar que las subclases decidan cual objeto instanciar. La clase Builder se encargará de instanciar y construir el objeto y retornará el objeto cuando esté construido completamente.

* + - Estructura:
    - Notas:
      * Tiene un objeto Factory construyendo un objeto incrementalmente usando una estructura compleja.
      * Se enfoca en construir objetos complejos paso a paso.
      * Regresa el producto como un paso final.
      * Ventajas:

Si hay un cambio en el objeto a instancias, solo afectará al Builder.

* + - * Desventajas:

Se tendrían muchos Builders para cada producto diferente del objeto.

* + Factory method
    - Problema:

Generar objetos pero la clase no puede anticipar de qué tipo serán.

* + - Contexto:

Los framework utilizar clases abstractas para definir y mantener las relaciones entre objetos. Su responsabilidad es crear los objetos.

* + - Solución:

Definir una **interfaz o clase padre** para crear un objeto y dejando la elección del tipo a las subclases, la creación se aplaza hasta el tiempo de ejecución.

* + - Notas:
      * Tiene un objeto Factory que produce objetos del mismo tipo pero características distintas implementadas en subclases dependiendo de lo que requiera el cliente. Ejemplo: Carro deportivo y carro familiar. El Factory retorna 2 carros pero con diferentes valores en sus atributos.
      * El producto se regresa inmediatamente.
  + Abstract method
    - Problema:

Se desea proporcionar una biblioteca o familias de clases de productos, revelando solo sus interfaces, no sus implementaciones.

* + - Contexto:

Evitar añadir código a las clases existentes para que encapsule información mas general.

* + - Solución:

Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

* + - Notas:
      * Tiene un objeto Factory que produce objetos del mismo tipo pero familia y características distintas implementadas en subclases dependiendo de lo que requiera el cliente. Ejemplo: Carro deportivo y carro familiar de diferente marca. El Factory retorna 2 carros pero con diferentes valores en sus atributos y de la fábrica que especifique el cliente.
      * Hace énfasis a una familia de objetos de producto.
      * El producto se regresa inmediatamente.
  + Singleton
    - Solo una instancia de un objeto se utiliza en toda la aplicación, asegurándose que solo sea una sola instancia. En Java es creando una clase con atributos privados y una función estática que retorna el mismo objeto inicializado.
* **Patrones de estructura** (interacción de clases y objetos)
  + Adapter
    - Problema:

Se desea utilizar una clase existente, pero su interfaz no coincide con la que se necesita.

Se desea crear clases reutilizables que cooperen con clases sin relación o imprevistas.

* + - Contexto:

Relacionar dos componentes que no tienen una interfaz común.

* + - Solución:

Convertir la interfaz de una clase en otra que el cliente espera.

* + - Notas:
      * Se debe crear una clase Adapter que sirva de unión entre las 2 clases, la nueva y la vieja, para reutilizar lo más que se pueda código de la clase antigua.
      * Otra forma es respetar la implementación de la creación de un objeto en el Adapter, como llamando la función igual que la de la clase anterior y dentro utilizar las funciones de la nueva clase, de esta manera el cliente no tendrá que cambiar su llamado al método y el funcionamiento será la de la nueva clase.
  + Facade
    - Problema:

Proveer una interfaz simple a un subsistema complejo.

* + - Contexto:

Minimizar la comunicación y dependencias entre subsistemas.

* + - Solución:

Proporcionar una interfaz unificada para un conjunto de interfaces de un subsistema.

* + - Notas:
      * El cliente invoca solo una función de la clase Facade y esta internamente une y realiza tareas de otras clases.
  + Decorator
    - Problema:

Agregar responsabilidades a objetos individuales dinámica y transparentemente.

* + - Contexto:

A veces queremos añadir responsabilidades a un objeto de forma dinámica.

* + - Solución:

Adjuntar responsabilidades adicionales a un objeto de forma dinámica.

Decorators proporciona una alternativa flexible de las subclases para ampliar la funcionalidad.

* + - Notas:
      * Se utiliza para aumentar la interfaz de cierta clase.
      * Permite cambiar el exterior de un objeto.
* **Patrones de comportamiento** (interacción entre objetos)
  + Command
    - Problema:

Especificar, encolar y ejecutar peticiones en diferentes tiempos.

Callbacks.

* + - Contexto:

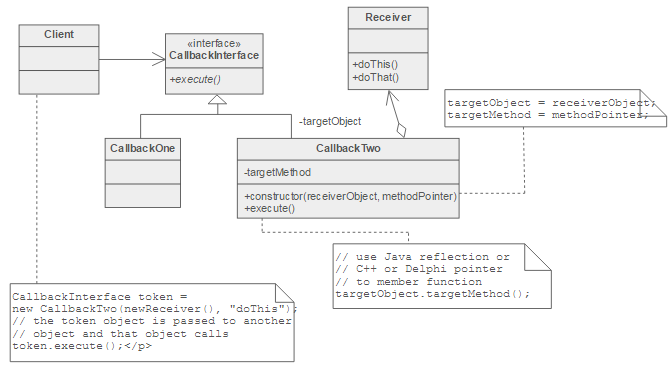
Emisión de peticiones a objetos sin saber nada de la operación que se solicite o el receptor de la solicitud.

* + - Solución:

Encapsular una petición como un objeto.

Almacenar las peticiones en una cola.

* + - Estructura:



* + - Ejemplo (Switch de la luz):
      * Se tiene una interface (Command) con un método abstracto execute().
      * Se tienen 2 clases (OnLight y OffLight) implementando el método execute() que manda llamar un método de otra clase (Light) con el proceso a realizar (prender la luz o apagar la luz).
      * Se tiene una clase que es la que interactuará con el cliente (LightSwitch), esta clase mandará llamar funciones (on u off) de otra clase (Switch) dependiendo del comando que el cliente quiera realizar.
      * La clase invoker (Switch) mandará llamar el execute del comando específico para la acción pedida.
  + Observer
    - Problema:

Un cambio en un objeto requiere de cambios en otros, y no se sabe cuántos objetos necesitan ser cambiados.

Una abstracción tiene dos aspectos dependientes.

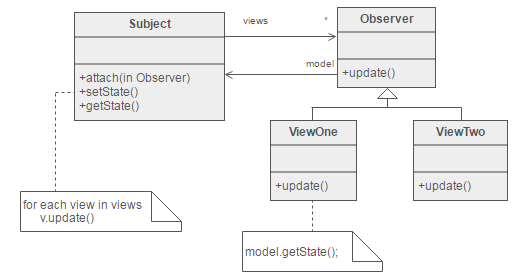
* + - Contexto:

Al particionar un sistema en una colección de clases cooperativas, se requiere mantener la consistencia entre objetos relacionados.

* + - Solución:

Definir una dependencia uno a muchos entre objetos, para que al cambiar un objeto, todos sus dependientes sean notificados automáticamente.

* + - Estructura:



* + - Ejemplo (Foro de discusión):
      * Se tiene una interface (Publisher) con los métodos abstractos addUser(User), removeUser(User) y notifyAll() para administrar los dependientes y notificarles cuando cambie un objeto.
      * Se tiene una clase (Forum) que implementa esta interface y también un método writePost(String) que es lo que detonará la notificación a todos los dependientes.
      * Se tiene una interface (Suscriber) con el método abstracto notifyUser(String) que deberá tener todos los dependientes y será el que se mandará llamar cuando un objeto cambie.
      * Se crean 2 clases (User1 y User2) que implementan la interface. La implementación del método es realizando las acciones que necesita hacer ese dependiente cuando algún otro dependiente cambie algo.
  + Strategy
    - Problema:

Requieres diferentes variantes de un algoritmo.

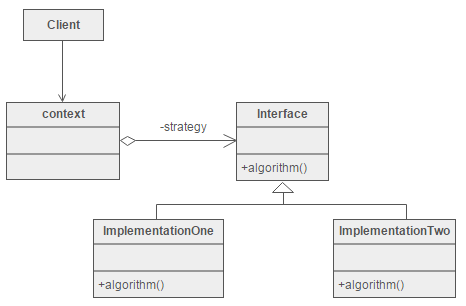
* + - Contexto:

Clases relacionadas solo difieren en su comportamiento.

* + - Solución:

Defina una familia de algoritmos, los encapsula y los hace intercambiables.

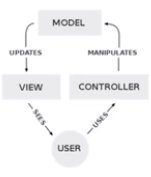
* + - Estructura:



* + - Notas:
      * Permite cambiar el interior.
      * Se utiliza una clase (interface) con función abstracta para que cada subclase que la implementa tenga diferente funcionalidad.
    - Ejemplo (Algoritmos de búsqueda):
      * Se tiene una interface (TextFinder) con un método abstracto find(String).
      * Se tienen 2 clases (Algoritmo1 y Algoritmo2) que implementan la interface y cada una tendrán implementado el método abstracto con diferentes algoritmos para búsqueda.

**PATRONES DE ARQUITECTURA:**

* MVC (Model View Controler)
  + Patrón para implementación de GUI.
  + Divide una aplicación de SW en 3 partes interconectadas (Model-View-Controler).



* + Implementación del patrón de Observer, el Controler notifica al Model y View de que algo cambió.
* EDA (Event-driven Architecture)
  + Promueve la producción, detección, consumo de y reacción a los eventos.
  + Se puede aplicar para el diseño e implementación de aplicación y sistemas que transmiten eventos entre los componentes de SW y servicios débilmente acoplados.