**Patrones de software:**

***"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a este problema, de tal ,modo que esta solución se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces"* Christopher Alexander.**

**Patrón de diseño a nivel sistema:**

**a) Estilo arquitectónico:** Solución de organización a nivel del sistema

-Expresa un esquema de organización estructural para sistemas de software.

-Provee un conjunto de tipos de elementos predefinidos,

-Especifica sus responsabilidades e incluye reglas y guías para organizar las relaciones entre ellos.

Son patrones de diseño de software que ofrecen soluciones a problemas de arquitectura de software en ingeniería de software. Dan una descripción de los elementos y el tipo de relación que tienen junto con un conjunto de restricciones sobre cómo pueden ser usados. Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. En comparación con los patrones de diseño, los patrones arquitectónicos tienen una nivel de abstracción mayor.

**Beneficios de usar este patrón:**

• Permite seleccionar una solución entendible y probada a ciertos problemas, definiendo los principios organizativos del sistema.

• Al basar la arquitectura en estilos que son conocidos las personas entienden más

fácilmente las características importantes de la misma.

Uno de los aspectos más importantes de los patrones arquitectónicos es que encarnan diferentes atributos de calidad. Por ejemplo, algunos patrones representan soluciones a problemas de rendimiento y otros pueden ser utilizados con éxito en sistemas de alta disponibilidad.

**Ejemplos de patrones arquitectónicos incluyen los siguientes:**

1) **Programación por capas**: El sistema se organiza en capas. Cada una provee un conjunto de servicios a las capas superiores y requiere servicios de las inferiores.

-Modelo estricto:

una capa sólo utiliza servicios

de la inmediata inferior.

-Modelo relajado

se pueden “saltar” capas.

-Definición de protocolos mediante los que interactúan las capas.

**Ventajas**

-Facilita la comprensión

-Facilita mantenimiento

-Facilita reutilización

-Facilita portabilidad

**Desventajas**

-No siempre es fácil estructurar en capas ni identificar los niveles de abstracción a partir de los Requerimientos.

- La interpretación de comandos en múltiples niveles puede afectar el desempeño

2) **Shared Data**: Hay un almacenamiento central de datos y un conjunto de componentes que operan sobre éste.

-Interacción - intercambio de datos persistentes

- Múltiples accesos a los datos

-Al menos un almacenamiento compartido

-Si se avisa al consumidor de nuevos datos de interés es un Pizarrón (Blackboard)

-Si es el consumidor quien busca los datos es un Repositorio

**Ventajas y desventajas:**

Forma eficiente de compartir grandes cantidades de datos.

-No se transmiten datos de un componente a otro.

-Sin embargo, los subsistemas deben estar de acuerdo en el modelo de datos del repositorio.

-Las componentes que producen datos no necesitan conocer como van a ser usados esos datos.

-Actividades como respaldos, seguridad, control de acceso y recuperación están centralizadas.

-Sin embargo, diferentes componentes podría tener distintas políticas de recuperación, respaldo, etc.

3) **Cliente-Servidor:**

El sistema se descompone en servicios y sus servidores asociados y en clientes que acceden y usan dichos servicios.

Estilo compuesto por:

-Servidores que ofrecen servicios.

-Clientes que usan servicios ofrecidos por los servidores.

-Una red que permite que los clientes accedan a los servidores.

-Esto no es estrictamente necesario ya que clientes y servidores

pueden estar en una misma máquina.

-Los clientes conocen a los servidores pero no a otros clientes y los servidores no tienen porqué conocer a los clientes

-la asignación de procesos a procesadores no tiene porqué ser 1:1

4) **Descomposición O.O:**

Se descompone el sistema en un conjunto de objetos que se comunican.

-Representación de datos y operaciones asociadas se encapsulan en un objeto.

-Herencia, polimorfismo, sobrecarga de operadores, enlace dinámico.

**Ventajas y Desventajas:**

**Ventajas**:

-La implementación de los objetos puede ser cambiada sin afectar a otros objetos.

Promueve la reutilización de componentes.

-Muchos objetos representan entidades de la realidad por lo que es fácil entender la estructura del sistema.

**Desventajas:**

-Para usar servicios se debe conocer el nombre de la interface de otro objeto.

-Los cambios en las interfaces afectan a todos los objetos que la usan

5) **Tubos y filtros:** Se descompone el sistema en módulos funcionales

-Interacción - sucesiva transformación de flujos de datos

-Los datos llegan a un filtro, se transforman y son pasados a través

de tubos al siguiente filtro

-Un único filtro puede pasar datos a múltiples tubos y recibir datos

de múltiples tubos

-Cada filtro es independiente del resto y no conoce la identidad de

los otros filtros

-La transformación del filtro puede comenzar antes de terminar de

leer la entrada

-Respetando el grafo, no importa la secuencia (paralelismo)

**Ventajas:**

-Se pueden reutilizar los filtros.

-Es intuitivo pensar en secuencias de procesamientos de datos.

-Agregar nuevas transformaciones de forma que el sistema evolucione es sencillo.

**Desventajas:**

-Acordar cuál es el formato de los datos.

-Tiene que ser “genérico” si las componentes son reusadas.

-Sistemas interactivos son difíciles de construir con este estilo.

-Ineficiente si hace más de lo que debe o si los filtros repiten chequeos

6) **Modelos de control:**

Modelos de control a nivel de la arquitectura se preocupan del flujo de control entre subsistemas

-Control centralizado: Un subsistema controla al resto

- Control basado en eventos: Cada subsistema puede responder a eventos externos

-Eventos generados por otros subsistemas

-Eventos generados por el ambiente del sistema

7) **Publicar- Suscribir:** Sistema de componentes independientes que anuncian y se suscriben a eventos

-Interacción vía eventos anunciados

-Los componentes se suscriben a eventos

-Es trabajo del run-time asegurarse que cada evento publicado sea entregado a todos los suscriptores de dicho evento

- Una forma es la invocación implícita

8) **Sistemas distribuidos:** El procesamiento de la información es distribuido entre varias computadoras.

**Ventajas:**

-Se comparten recursos.

-Apertura. Normalmente estos sistemas están diseñados con protocolos estándar.

-Concurrencia.

-Escalabilidad

-Tolerancia a fallas

**Desventajas:**

-Complejidad

-Seguridad

-Difíciles de gestionar

-Muy poco predecibles

Ejemplos

-Cliente-servidor

-Objetos distribuidos

-Peer-to-peer

-Service oriented architecture (SOA)

9) **Orientada a servicios:** Los servicios se basan en estándares basados en XML. Entonces, pueden funcionar en cualquier plataforma y ser escritos en cualquier lenguaje.

Estándares más conocidos:

-SOAP - Simple Object Access Protocol

-WSDL - Web Services Description Language

-UDDI - Universal Description, Discovery and

Integration.

**Evaluación de Arquitecturas:**

-Cambiar la arquitectura de un producto ya construido requiere mucho esfuerzo.

Entonces, es importante evaluar la arquitectura antes de implementarla completamente

**-Verificar los requisitos de calidad establecidos**

-Evaluaciones a posteriori resultan útiles como forma de aprendizaje y estudio de posibilidades de mejora, por ej. para una nueva versión del producto

**-Software Engineering Institute (SEI) propone:**

-Architecture Tradeoff Analysis Method (ATAM)

-Software Architecture Analysis Method (SAAM)

**b) Patrón de diseño:** Aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.

Propósito: Compartir una solución probada, ampliamente aplicable a un problema particular de diseño.

El patrón se presenta en una forma estándar que permite que sea fácilmente reutilizado.

**Piezas importantes de un patrón:**

Su nombre, el problema (cuando aplicar un patrón), la solución (descripción abstracta del problema) y las consecuencias (costos y beneficios).

**Un patrón de diseño debe cumplir al menos con los siguientes objetivos**

1. Estandarizar el lenguaje entre programadores
2. Evitar perder tiempo en soluciones a problemas ya resueltos o conocidos
3. Crear código reusable (excelente ventaja)

### Creacionales:

Corresponden a patrones de diseño software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstraer dicha creación:

[**Object Pool**](https://es.wikipedia.org/wiki/Object_Pool_%28patr%C3%B3n_de_dise%C3%B1o%29)(no pertenece a los patrones especificados por GoF): se obtienen objetos nuevos a través de la clonación. Utilizado cuando el costo de crear una clase es mayor que el de clonarla. Especialmente con objetos muy complejos. Se especifica un tipo de objeto a crear y se utiliza una interfaz del prototipo para crear un nuevo objeto por clonación. El proceso de clonación se inicia instanciando un tipo de objeto de la clase que queremos clonar.

[**Abstract Factory**](https://es.wikipedia.org/wiki/Abstract_Factory_%28patr%C3%B3n_de_dise%C3%B1o%29)(fábrica abstracta): permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando. El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).

[**Builder**](https://es.wikipedia.org/wiki/Builder_%28patr%C3%B3n_de_dise%C3%B1o%29) (constructor virtual): abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.

[**Factory Method**](https://es.wikipedia.org/wiki/Factory_Method_%28patr%C3%B3n_de_dise%C3%B1o%29) (método de fabricación): centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística, es decir, la diversidad de casos particulares que se pueden prever, para elegir el subtipo que crear. Parte del principio de que las subclases determinan la clase a implementar.

**Prototype** (prototipo): crea nuevos objetos clonándolos de una instancia ya existente.

[**Singleton**](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o_Singleton) (instancia única): garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia. Restringe la instanciación de una clase o valor de un tipo a un solo objeto.

**Estructurales:**

Su nombre es muy descriptivo, se ocupa de resolver problemas sobre la estructura de las clases. Por ejemplo:

* **Bridge** (Puente) Separa la abstracción de la implementación.
* **Decorator** (Decorador) Agrega funcionalidades a una clase de forma dinámica.
* **Facade** (Fachada) Nos provee una interfaz unificada y simple para acceder a un sistema más complejo.
* [**Adapter o Wrapper**](https://es.wikipedia.org/wiki/Adapter_%28patr%C3%B3n_de_dise%C3%B1o%29) (Adaptador o Envoltorio): Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
* [**Composite**](https://es.wikipedia.org/wiki/Composite_%28patr%C3%B3n_de_dise%C3%B1o%29) (Objeto compuesto): Permite tratar objetos compuestos como si de uno simple se tratase.
* [**Flyweight**](https://es.wikipedia.org/wiki/Flyweight_%28patr%C3%B3n_de_dise%C3%B1o%29)(Peso ligero): Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
* [**Proxy**](https://es.wikipedia.org/wiki/Proxy_%28patr%C3%B3n_de_dise%C3%B1o%29)**:** Mantiene un representante de un objeto.
* [**Módulo**](https://es.wikipedia.org/wiki/M%C3%B3dulo_%28patr%C3%B3n_de_dise%C3%B1o%29)**:** Agrupa varios elementos relacionados, como clases, singletons, y métodos, utilizados globalmente, en una entidad única.

**De comportamiento**:

Nos ayuda a resolver problemas relacionados con el comportamiento de la aplicación. Ofrece soluciones respecto a la interacción y responsabilidad entre objetos y clases. A nivel programación.

Por ejemplo:

Se definen como patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan:

[**Chain of Responsibility**](https://es.wikipedia.org/wiki/Chain_of_Responsibility_%28patr%C3%B3n_de_dise%C3%B1o%29) (Cadena de responsabilidad): Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.

[**Command**](https://es.wikipedia.org/wiki/Command_%28patr%C3%B3n_de_dise%C3%B1o%29) (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.

[**Interpreter**](https://es.wikipedia.org/wiki/Interpreter_%28patr%C3%B3n_de_dise%C3%B1o%29) (Intérprete): Dado un lenguaje, define una gramática para dicho lenguaje, así como las herramientas necesarias para interpretarlo.

[**Iterator**](https://es.wikipedia.org/wiki/Iterator_%28patr%C3%B3n_de_dise%C3%B1o%29)(Iterador): Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.

[**Mediator**](https://es.wikipedia.org/wiki/Mediator_%28patr%C3%B3n_de_dise%C3%B1o%29) (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.

[**Memento**](https://es.wikipedia.org/wiki/Memento_%28patr%C3%B3n_de_dise%C3%B1o%29)(Recuerdo): Permite volver a estados anteriores del sistema.

[**Observer**](https://es.wikipedia.org/wiki/Observer_%28patr%C3%B3n_de_dise%C3%B1o%29)(Observador): Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.

[**State**](https://es.wikipedia.org/wiki/State_%28patr%C3%B3n_de_dise%C3%B1o%29)(Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

[**Strategy**](https://es.wikipedia.org/wiki/Strategy_%28patr%C3%B3n_de_dise%C3%B1o%29)(Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.

[**Template Method**](https://es.wikipedia.org/wiki/Template_Method_%28patr%C3%B3n_de_dise%C3%B1o%29)(Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

[**Visitor**](https://es.wikipedia.org/wiki/Visitor_%28patr%C3%B3n_de_dise%C3%B1o%29)(Visitante): Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

**MCV:** Model View Controller:Modelo Vista Controlador: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Envía a la 'vista' aquella parte de la información que en cada momento se le solicita para que sea mostrada (típicamente a un usuario). Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.

El Controlador (controller): Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo' (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos), por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'

La Vista: Presenta el 'modelo' (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz del usuario) por lo tanto requiere de dicho 'modelo' la información que debe representar como salida.

**PATRONES DE ARQUITECTURA VS. PATRONES DE DISEÑO**

Si al aplicar el patrón: si este es relevante a la totalidad del sistema entonces hablamos de un patrón de arquitectura; en cambio, si este sólo concierne a un subcomponente, nos referimos a un patrón de diseño.

Tomen como caso el patrón de Layers, este es claramente un patrón arquitectónico, ya que concierne al diseño general de la aplicación. Mientras que el patrón Active Record, que lidia con los mecanismos de persistencia de datos es un patrón de diseño.

Pero, qué pasa cuando lo que era totalidad se vuelve un subcomponente?. Cambiarían entonces sus patrones de arquitectura a diseño?. Aquí es donde no es tan fácil la respuesta.

Hay otros patrones que solapan responsabilidades de todo-parte, por ejemplo el MVC . Según como se aplique puede ser un patrón de diseño o arquitectura.