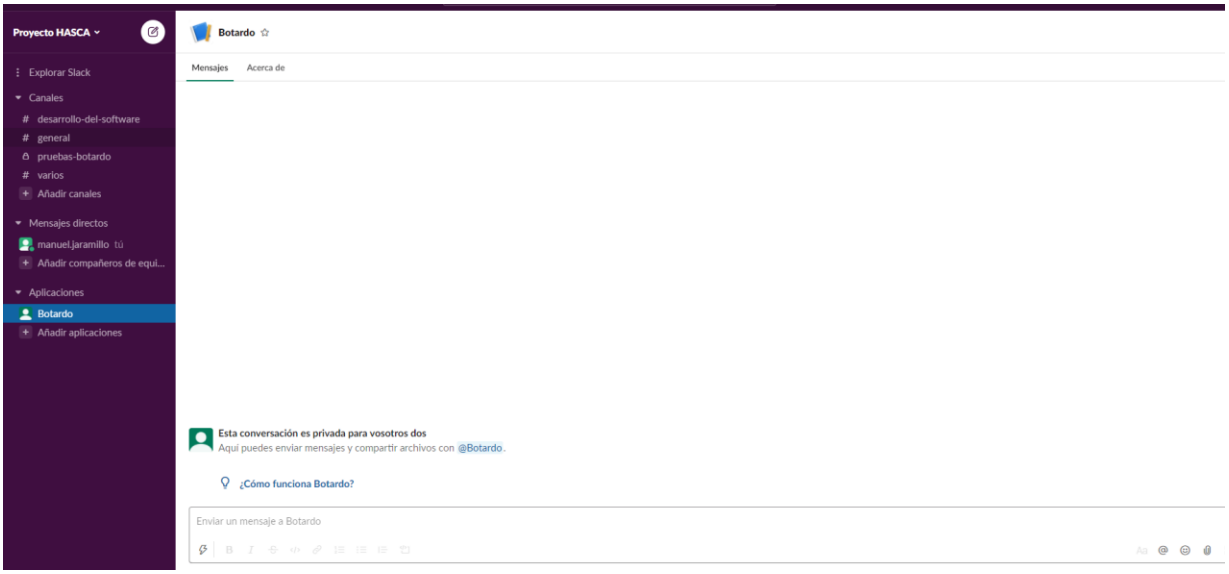
# Slack

https://app.slack.com/client/T01C4D9MGCE/C01C4DBAVC6



# Atributos de calidad / Requisitos no funcionales

1. ¿Qué es la calidad de software?

Un software es de calidad no solo cuando cumple correctamente la funcionalidad requerida y le aporta valor al cliente que lo usa, además, cuando su coste de mantenimiento es bajo y la dificultad para introducir nuevos cambios (nuevos requisitos) es baja.
Se debe escribir código pensando en que sufrirá cambios necesariamente, esto determinará el coste de mantenerlo e incluso el éxito o fracaso del proyecto.
También se debe tener en cuenta que se debe hacer funcionar un negocio, y pocos negocios son estáticos por lo que deben cambiar, optimizar o mejorar continuamente.

2. ¿Cuáles son los atributos de calidad de un software?

Los atributos de calidad son características no funcionales que se consideran deseables en un sistema de software. No es necesario usar todos los atributos, algunas serán más importantes que otras dependiendo el sistema.

**Simplicidad**

Esta es la ausencia de complejidad o dificultades. En el desarrollo de software es bueno diferenciar entre complejidades esenciales y accidentales.

- **Complejidad esencial:** Las que son propias del problema a solucionar.
- **Complejidades accidentales:** Aquellas que surgen de malas decisiones del diseño.

Diferenciar de estas nos permite atacar las dificultades accidentales, buscando soluciones simples.

**Correctitud, consistencia, completitud**

Correctitud: Ausencia de errores.
Consistencia: Coherencia entre las operaciones que realiza el usuario.
Completitud: Capacidad del sistema para realizar las operaciones que el usuario podría requerir.

**Robustez**

Cuando un sistema goza de buena salud y garantiza que seguirá así. Lo que caracteriza un sistema robusto es:

- La capacidad de ser modificado sin introducir errores.
- La durabilidad del sistema funciona correctamente.

**Flexibilidad**

Capacidad para admitir cambios que pueden ser necesarios por un cambio de requerimiento, como también la detección de un error que debe ser corregido. La extensibilidad, es la posibilidad de agregar nuevos requerimientos.

**Performance**

Medida de eficiencia en el uso de recursos del sistema en ejecución, como por ejemplo el uso de procesador, memoria, etc.

**Escalabilidad**

Capacidad de un sistema para trabajar con diferentes cantidades de trabajo, como cambios en el volumen de datos o flujos de pedidos.
Existe la escalabilidad de un sistema hacia arriba, donde se mide la capacidad del sistema para manejar, por ejemplo, un mayor número de datos, mientras que la escalabilidad hacia abajo es la posibilidad de un sistema de adaptarse a un entorno más sencillo. Un error es confundir escalabilidad con extensibilidad.

**Seguridad**

Comprobar la identidad de las personas que intentan acceder al sistema.

**Usabilidad**

La facilidad con la que el sistema se puede utilizar o bien aprender a utilizar.

**Constructibilidad**

Medida inversa a la complejidad de la construcción del sistema. Las decisiones de diseño pueden afectar severamente la dificultad para construir ese sistema.

3. ¿Qué es un requisito no funcional? ¿Cuál es la diferencia con un requisito funcional?

Los requisitos no funcionales son aquellos que elaboran la característica de rendimiento del sistema y definen las restricciones sobre cómo lo hará el sistema, mientras que los requisitos funcionales son aquellos requisitos que se ocupan de lo que el sistema debe hacer o proporcionar a los usuarios.

4. ¿Es deseable cumplir con todos los atributos de calidad? ¿Por qué?

No todos los sistemas de software deben tener en cuenta todos estos atributos o cualidades, algunas serán más importantes que otras dependiendo del sistema, y ciertamente no se pueden maximizar todas a la vez.

5. ¿Cuál es el rol del arquitecto de software?

El arquitecto de software debe ser una persona con amplios conocimientos técnicos, gran experiencia en programación, liderazgo y que ejerza las siguientes funciones:

- Gestión de los requisitos no funcionales y definición de la arquitectura de software.
- Selección de la tecnología.
- Mejora continua de la arquitectura.
- Facilitador.
- Líder y Formador.
- Aseguramiento de la calidad.

El desacoplamiento se refiere a la acción de quitar un módulo de software y que esté último no se vea afectado al momento de hacerlo y menos de cambiar a otro.
Los atributos de calidad que están relacionados a esto serían:

- robustez, ya que tiene la capacidad de cambiar piezas de un software sin errores.
- Flexibilidad, ya que puede admitir cambios por cambio de requerimiento o por corregir un error. Además de la posibilidad de agregar nuevos requerimientos al sistema.
- Escalabilidad, puede ocurrir que el sistema quiera volverse más grande aceptando nuevos volúmenes de datos o de flujo de pedidos.

*¿Qué es un estilo arquitectónico? ¿Cuáles son los principales estilos arquitectónicos?*
Un estilo arquitectónico describe una clase de arquitectura o piezas de la misma, se pueden reutilizar en situaciones semejantes a futuro. Desde que surgió la arquitectura de software se generalizó la arquitectura cliente servidor, luego entró en auge las arquitecturas en capas y las basadas en componentes más recientemente basadas en recursos y servicios. Se establece que una arquitectura se define mediante la afirmación:

Arquitectura del Software = Elementos, forma y razón.
Elementos:

- procesamiento: Suministra la transformación de los datos
- datos: Contiene la información a procesar
- conexión: Llamadas a procedimientos, mensajes, etc.

Forma: las propiedades y relaciones entre los elementos.
Razón: motivación para la elección de elementos y sus formas.

Entonces un estilo arquitectónico es un conjunto de reglas de diseño que identifica las clases de componentes y conectores que se pueden utilizar para componer el sistema o subsistema junto con las restricciones locales o globales de la forma en que la composición se lleva a cabo.

- Componentes: sistemas encapsulados indicando la forma de empaquetado y la forma en que interactúan con otros componentes.

- Conectores: los procesos interactúan por medio de protocolos de transferencia de mensajes o por flujo de datos, etc.

Los principales estilos son:

- Arquitectura centrada en datos
- Arquitectura de flujo de datos
- Arquitectura de llamada y retorno
- Arquitecturas orientadas a objetos
- Arquitecturas orientadas a aspecto
- Arquitectura orientada a servicios (SOA)
- Arquitectura estratificadas.

*¿Qué estilo arquitectónico es preferible si se quiere optimizar la mantenibilidad?*

La arquitectura más recomendada en mi opinión es microservicios, ya que divide los sistemas en partes individuales, permitiendo que se puedan tratar y abordar los problemas de manera independiente sin afectar al resto.

La arquitectura de microservicios es un método de desarrollo de aplicaciones software que funciona como un conjunto de pequeños servicios que se ejecutan de manera independiente y autónoma, proporcionando una funcionalidad de negocio completa. En ella, cada microservicio es un código que puede estar en un lenguaje de programación diferente, y que desempeña una función específica. Los microservicios se comunican entre sí a través de APIs, y cuentan con sistemas de almacenamiento propios, lo que evita la sobrecarga y caída de la aplicación.

Los microservicios han creado infraestructuras IT más adaptables y flexibles. Porque si se quiere modificar solamente un servicio, no es necesario alterar el resto de la infraestructura.

# OAuth2

1. ¿Qué es?

OAuth 2.0 es un estándar abierto para la autorización de APIs, que nos permite compartir información entre sitios sin tener que compartir la identidad.

2. ¿Quién lo utiliza?

Es un mecanismo utilizado hoy en día por grandes compañías como Google, Facebook, Microsoft, Twitter, GitHub o LinkedIn, entre otros.

3. ¿Por qué se utiliza?

Se utiliza para disminuir la necesidad que se establece del **envío continuo de credenciales entre cliente y servidor.**

Donde más fuerza toma es en la integración con aplicaciones a terceros, ya que con OAuth2 el usuario delega la capacidad de realizar ciertas acciones, no todas, a las cuales da su consentimiento para hacerlas a su nombre.

4. ¿Cuáles son sus limitaciones?

El principal inconveniente es que este estándar no es infalible, por lo que algunos actores de amenazas podrían aprovecharse de algún usuario para obtener amplios accesos a sus cuentas en línea e incluso robar sus credenciales de acceso, exponiéndose a otro tipo de ataques.
Cover Redirect Vulnerability es una redirección no validada. Esta vulnerabilidad consiste en que ciertos sitios web, que implementan OAuth 2.0 como mecanismo de autenticación, aceptan parámetros dentro de la URL que provocan una redirección de los usuarios a otros sitios web sin la correcta validación de estos, incluso a dominios diferentes. El problema está en que en esta redirección se incluyen datos tan sensibles como el token o código de acceso que utiliza OAuth 2.0 para acceder a la información de los usuarios, por lo que no solamente presenta implicaciones sobre la seguridad, sino que también en el cumplimiento de la LOPD (LEY DE PROTECCIÓN DE DATOS).

1. ¿Cuál es la diferencia entre una Imagen Docker y un Contenedor Docker?

La imagen es la plantilla que se utilizará por una ejecución, que en este caso es el contenedor, y este último es una instancia de ejecución que encapsula la arquitectura de software funcionando.

2. ¿Cuál es la diferencia entre el archivo Dockerfile y el archivo Docker-compose.yml?

Docker.compose es el que se encarga de definir la estructura de las tecnologías distribuyendo la imágen, el volumen y la manera de ser ejecutados. Dockerfile es un archivo de comando que instancia instrucciones de variables de ambiente y de ejecución (Ej: pip, npm, install, etc.)

3. ¿Si utilizo un contenedor que contiene una Base de Datos (Redis, Mysql u otro), cómo y dónde se guardan los datos?

Los datos serán almacenados en un volumen creado por el contenedor, por ejemplo, redis-data:/data, ya que docker no almacena datos.

4. ¿Qué es un port? ¿Por qué algunas imágenes requieren hacer un bind entre distintos puertos?

Port se refiere a un puerto del pc, y algunas imágenes necesitan hacer un bind para poder comunicarse con la máquina que lo está ejecutando.

# Trabajo autónomo 2

**¿Qué son los principios SOLID?**

Los 5 principios SOLID de diseño de aplicaciones de software son:
S – Single Responsibility Principle (SRP)
O – Open/Closed Principle (OCP)
L – Liskov Substitution Principle (LSP)
I – Interface Segregation Principle (ISP)
D – Dependency Inversion Principle (DIP)

*Principio de Responsabilidad Única*

"A class should have one, and only one, reason to change."

La S del acrónimo del que hablamos hoy se refiere a Single Responsibility Principle (SRP). Según este principio "una clase debería tener una, y solo una, razón para cambiar".
El principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

*Principio de Abierto/Cerrado*

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro "Object Oriented Software Construction" y dice: "Deberías ser capaz de extender el comportamiento de una clase, sin modificarla". En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.

*Principio de Sustitución de Liskov*

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que "las clases derivadas deben poder sustituirse por sus clases base".
Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

*Principio de Segregación de la Interfaz*

En el cuarto principio de SOLID, el tío Bob sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para una finalidad concreta.
En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces que definen pocos métodos que tener una interfaz forzada a implementar muchos métodos a los que no dará uso.

*Principio de Inversión de Dependencias*
Llegamos al último principio: “Depende de abstracciones, no de clases concretas”. Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

**¿Cuáles son los argumentos en favor y en contra de SOLID?**

Aunque estos cinco principios son considerados por muchos como una base fundamental de un buen desarrollo o al menos como una guía a tener en cuenta, no son pocos los profesionales que critican los principios SOLID.

Los acusan de ambiguos, confusos, de complicar el código, de demorar el proceso de desarrollo y los tildan incluso de totalmente equivocados e innecesarios.

**¿A qué requisitos no funcionales responden? ¿A qué requisitos no funcionales no responden?**

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable. Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
Permitir escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.
En definitiva, desarrollar un software de calidad.

En este sentido la aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de patrones de diseño, que nos permitirán mantener una alta cohesión y, por tanto, un bajo acoplamiento de software.

# Patrones de diseño

**¿Qué son los patrones de diseño de software?**

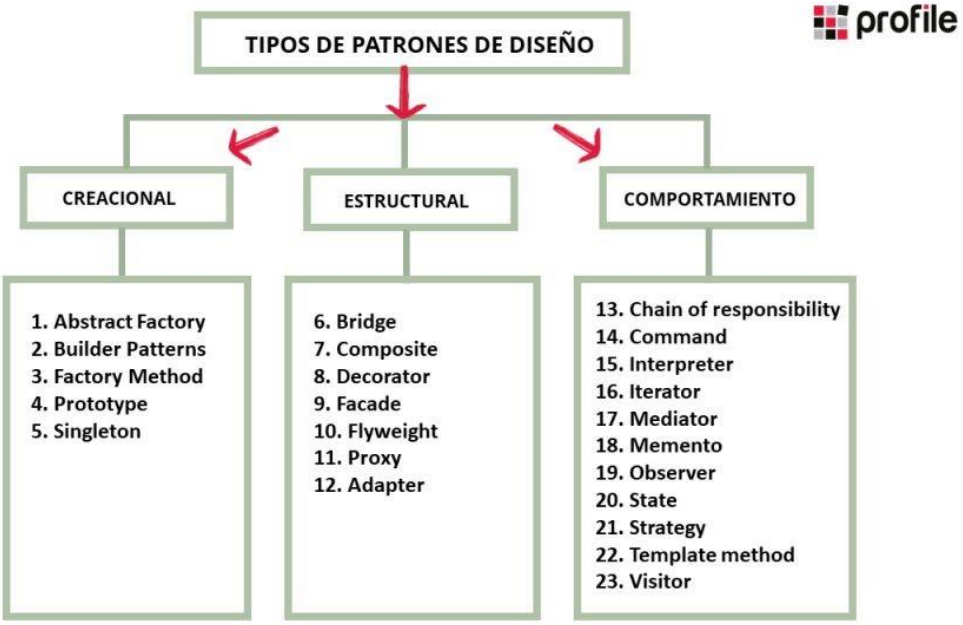
Es una solución general y reutilizable aplicable a diferentes problemas de diseño de software. Se trata de plantillas que identifican problemas en el sistema y proporciona soluciones apropiadas a problemas generales a los que se han enfrentado los desarrolladores durante un largo periodo de tiempo, a través de prueba y error.

**¿Por qué usar patrones de diseño?**

El gran crecimiento del sector de las tecnologías de la información ha hecho que las prácticas de desarrollo de software evolucionen. Antes se requería completar todo el software antes de realizar pruebas, lo que suponía encontrarse con problemas. Para ahorrar tiempo y evitar volver a la etapa de desarrollo una vez que este ha finalizado, se introdujo una práctica de prueba durante la fase de desarrollo. Esta práctica se usa para identificar condiciones de error y problemas en el código que pueden no ser evidentes en ese momento. En definitiva, los patrones de diseño te ayudan a estar seguro de la validez de tu código, ya que son soluciones que funcionan y han sido probados por muchísimos desarrolladores siendo menos propensos a errores.

**Tipos de patrones de diseño**

Los patrones de diseño más utilizados se clasifican en tres categorías principales, cada patrón de diseño individual conforma un total de 23 patrones de diseño. Las cuatro categorías principales son:



**Patrones creacionales**

Los patrones de creación proporcionan diversos mecanismos de creación de objetos, que aumentan la flexibilidad y la reutilización del código existente de una manera adecuada a la situación. Esto le da al programa más flexibilidad para decidir qué objetos deben crearse para un caso de uso dado.

Estos son los patrones creacionales:

- Abstract Factory

En este patrón, una interfaz crea conjuntos o familias de objetos relacionados sin especificar el nombre de la clase.

- Builder Patterns

Permite producir diferentes tipos y representaciones de un objeto utilizando el mismo código de construcción. Se utiliza para la creación etapa por etapa de un objeto complejo combinando objetos simples. La creación final de objetos depende de las etapas del proceso creativo, pero es independiente de otros objetos.

- Factory Method

Proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán. Proporciona instanciación de objetos implícita a través de interfaces comunes.

- Prototype

Permite copiar objetos existentes sin hacer que su código dependa de sus clases. Se utiliza para restringir las operaciones de memoria / base de datos manteniendo la modificación al mínimo utilizando copias de objetos.

- Singleton

Este patrón de diseño restringe la creación de instancias de una clase a un único objeto.

**Patrones estructurales**

Facilitan soluciones y estándares eficientes con respecto a las composiciones de clase y las estructuras de objetos. El concepto de herencia se utiliza para componer interfaces y definir formas de componer objetos para obtener nuevas funcionalidades.

- Adapter

Se utiliza para vincular dos interfaces que no son compatibles y utilizan sus funcionalidades. El adaptador permite que las clases trabajen juntas de otra manera que no podrían al ser interfaces incompatibles.

- Bridge

En este patrón hay una alteración estructural en las clases principales y de implementador de interfaz sin tener ningún efecto entre ellas. Estas dos clases pueden desarrollarse de manera independiente y solo se conectan utilizando una interfaz como puente.

- Composite

Se usa para agrupar objetos como un solo objeto. Permite componer objetos en estructuras de árbol y luego trabajar con estas estructuras como si fueran objetos individuales.

- Decorator

Este patrón restringe la alteración de la estructura del objeto mientras se le agrega una nueva funcionalidad. La clase inicial permanece inalterada mientras que una clase decorator proporciona capacidades adicionales.

- Facade

Proporciona una interfaz simplificada para una biblioteca, un marco o cualquier otro conjunto complejo de clases.

- Flyweight

El patrón Flyweight se usa para reducir el uso de memoria y mejorar el rendimiento al reducir la creación de objetos. El patrón busca objetos similares que ya existen para reutilizarlo en lugar de crear otros nuevos que sean similares.

- Proxy

Se utiliza para crear objetos que pueden representar funciones de otras clases u objetos y la interfaz se utiliza para acceder a estas funcionalidades.

**Patrones de comportamiento**

El patrón de comportamiento se ocupa de la comunicación entre objetos de clase. Se utilizan para detectar la presencia de patrones de comunicación ya presentes y pueden manipular estos patrones.

Estos patrones de diseño están específicamente relacionados con la comunicación entre objetos.

- Chain of responsibility

El patrón de diseño Chain of Responsibility es un patrón de comportamiento que evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.

- Command

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación permite parametrizar métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y respaldar operaciones que no se pueden deshacer.

- Interpreter

Se utiliza para evaluar el lenguaje o la expresión al crear una interfaz que indique el contexto para la interpretación.

- Iterator

Su utilidad es proporcionar acceso secuencial a un número de elementos presentes dentro de un objeto de colección sin realizar ningún intercambio de información relevante.

- Mediator

Este patrón proporciona una comunicación fácil a través de su clase que permite la comunicación para varias clases.

- Memento

El patrón Memento permite recorrer elementos de una colección sin exponer su representación subyacente

- Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que está siendo observado.

- State

En el patrón state, el comportamiento de una clase varía con su estado y, por lo tanto, está representado por el objeto de contexto.

- Strategy

Permite definir una familia de algoritmos, poner cada uno de ellos en una clase separada y hacer que sus objetos sean intercambiables.

- Template method

Se usa con componentes que tienen similitud donde se puede implementar una plantilla del código para probar ambos componentes. El código se puede cambiar con pequeñas modificaciones.

- Visitor

El propósito de un patrón Visitor es definir una nueva operación sin introducir las modificaciones a una estructura de objeto existente.