**Introducción a la calidad del software**

Se refiere a la calidad del software para cumplir con los requisitos, expectativas y necesidades establecidas por los usuarios de manera efectiva y confiable.

**3 pilares fundamentales de la calidad: Alcance (característica, funcionalidad), coste (recursos, presupuesto) y tiempo (planificación)**

**Estándares de calidad de software**

**Estándar ISO/IEC 25010:** es un estándar internacional que proporciona un marco para evaluar la calidad del software. Se enfoca en las características de calidad del software y proporciona una guía para medir y evaluar aspectos clave del software desde la perspectiva del usuario y las partes interesadas.

**Atributos de calidad:**

* **Funcionalidad:** El software hace lo que tiene y/o debe de hacer.
* **Fiabilidad:** Mide la capacidad del software para funcionar correctamente en condiciones específicas y durante un tiempo determinado, sin fallos.
* **Usabilidad:** El software es fácilmente usable por los usuarios.
* **Eficiencia:** El software hace uso óptimo de los recursos.
* **Mantenibilidad:** El software es fácilmente modificado, corregido y mejorado sin
* causas de efectos secundarios.
* **Portabilidad:** El software puede ser utilizado en diferentes entornos sin grandes cambios.
* **Seguridad:** Evalúa la capacidad del software para protegerse contra accesos no autorizados, ataques, y pérdida de datos.
* **Compatibilidad:** Define la capacidad del software para funcionar adecuadamente en conjunto con otros sistemas o componentes.
* **Confiabilidad:** El software opera de manera consistente, sin fallos ni pérdida de datos, asegurando la fiabilidad y la precisión de la información procesada.

**Estándar de calidad CMMI:** Es un modelo de madurez y calidad de procesos que se utiliza para mejorar el rendimiento en el desarrollo de software. Este modelo proporciona un conjunto de prácticas para mejorar los procesos de desarrollo.

**CMMI** se divide en cinco niveles de madurez, que representan etapas evolutivas en la mejora continua de los

procesos organizacionales:

* Nivel 1 - Inicial: Los procesos están poco definidos y son impredecibles. El éxito depende de los esfuerzos individuales y heroicos.
* Nivel 2 - Gestionado: Se establecen procesos básicos para el proyecto y se planifican y controlan las actividades.
* Nivel 3 - Definido: Los procesos se estandarizan y se documentan a nivel organizacional, lo que permite una ejecución más consistente.
* Nivel 4 - Cuantitativamente gestionado: Se establecen objetivos cuantitativos para los procesos y se utilizan métricas para medir el rendimiento y tomar decisiones basadas en datos.
* Nivel 5 - Optimizado: La organización se enfoca en la mejora continua de procesos basada en la

retroalimentación y la innovación.

**¿En que se diferencia el estándar ISO y el CMMI?**

Juntos, estos estándares proporcionan una combinación poderosa para mejorar la calidad del software de manera integral. ISO/IEC 25010 permite evaluar el producto final, mientras que CMMI ayuda a mejorar la eficacia y eficiencia de los procesos que llevan a ese producto.

**Principios SOLID**

Los principios SOLID son un conjunto de cinco principios fundamentales para el diseño de software orientado a objetos, propuestos por Robert C. Martin. Estos principios ayudan a crear software más flexible, fácil de mantener y escalable. Cada uno de los principios SOLID promueve buenas prácticas en el diseño de clases, funciones y módulos. A continuación, te los explico:

**S** - **(SRP) Principio de Responsabilidad Única**:

Cada clase debe tener una única responsabilidad o razón de cambio. Es decir, una clase debe estar encargada de cumplir solo una función específica dentro del sistema.

* **Ejemplo**: Una clase que maneje las operaciones de un carrito de compras no debería encargarse también de la facturación, ya que son responsabilidades diferentes.

**O** - **(OCP) Principio Abierto/Cerrado**:

Las entidades de software (clases, módulos, funciones) deben estar **abiertas para extensión**, pero **cerradas para modificación**. Esto significa que se debe poder extender el comportamiento de una clase sin modificar su código fuente.

* **Ejemplo**: Si quieres añadir una nueva funcionalidad a un sistema, lo ideal es crear una nueva clase o módulo en lugar de modificar una clase existente.

**L** - **(LSP) Principio de Sustitución de Liskov**:

Las clases derivadas deben ser reemplazables por sus clases base sin alterar el correcto funcionamiento del sistema. En otras palabras, los objetos de una subclase deben poder ser utilizados en lugar de objetos de la clase base sin romper la lógica del programa.

* **Ejemplo**: Si tienes una clase base Animal con un método caminar (), todas las subclases como Perro o Gato deben implementar este método de forma coherente.

**I** - **(ISP) Principio de Segregación de Interfaces**:

Los clientes no deberían estar obligados a depender de interfaces que no usan. Es preferible tener varias interfaces pequeñas y específicas a una interfaz grande que obligue a implementar métodos innecesarios.

* **Ejemplo**: Si tienes una interfaz Vehículo, en lugar de tener un método navegar () que no todos los vehículos usarán, podrías dividir esa interfaz en varias más específicas como Vehículo Terrestre, Vehículo Acuático, etc.

**D** - **(DIP) Principio de Inversión de Dependencias**:

Los módulos de alto nivel no deberían depender de módulos de bajo nivel; ambos deberían depender de abstracciones. Además, las abstracciones no deberían depender de detalles, sino que los detalles deben depender de abstracciones. Esto ayuda a desacoplar las diferentes partes del software.

* **Ejemplo**: En lugar de que una clase de alto nivel directamente cree instancias de clases concretas, estas dependencias deberían inyectarse desde el exterior (a través de la inversión de control o inyección de dependencias).

**Cohesión y acoplamiento**

**Cohesión** y **acoplamiento** son conceptos clave en el diseño de software que se refieren a la organización y relación entre los módulos o componentes de un sistema. Estos términos ayudan a evaluar la calidad del diseño de un sistema, favoreciendo la creación de software más fácil de mantener, entender y modificar.

**Cohesión:** Se refiere al grado en que los elementos dentro de un módulo (como funciones, métodos o clases) están relacionados entre sí y trabajan juntos para cumplir una única tarea o responsabilidad. Un módulo con alta cohesión tiene componentes que están muy bien integrados y colaboran directamente para lograr un objetivo específico.

**Tipos de cohesión**: Existen diferentes niveles de cohesión. La alta cohesión es deseable, mientras que la baja cohesión tiende a generar confusión y dificultad en el mantenimiento del sistema.

* **Alta cohesión**: Todas las funciones y datos dentro de un módulo están directamente relacionadas y se enfocan en una única tarea.
* **Baja cohesión**: Un módulo realiza tareas no relacionadas o contiene funcionalidades dispersas que no están claramente enfocadas en una responsabilidad.

**Ejemplo**:

* Alta cohesión: Una clase Usuario con métodos como registrarse (), iniciar Sesión () y cerrar Sesión () muestra alta cohesión porque todas las operaciones están relacionadas con la gestión de un usuario.
* Baja cohesión: Una clase Miscelánea que incluye métodos para calcular impuestos, enviar correos electrónicos y gestionar bases de datos tiene baja cohesión, ya que sus funciones no están claramente relacionadas.

**Acoplamiento:** Se refiere al grado de dependencia entre módulos o componentes de un sistema. Un sistema con bajo acoplamiento tiene módulos que son independientes entre sí, lo que facilita el mantenimiento y la evolución del software. Por el contrario, un alto acoplamiento significa que los módulos están fuertemente conectados, lo que puede dificultar los cambios y aumentar la probabilidad de errores al modificar el sistema.

**Tipos de acoplamiento**:

* **Bajo acoplamiento**: Los módulos interactúan entre sí a través de interfaces bien definidas y no dependen de los detalles internos del otro. Esto es deseable porque permite cambios en un módulo sin afectar a otros.
* **Alto acoplamiento**: Los módulos dependen de los detalles internos de otros módulos. Esto es problemático porque cualquier cambio en un módulo puede requerir modificaciones en varios otros módulos.

**Ejemplo**:

* Bajo acoplamiento: Un módulo Base de datos que solo expone métodos públicos como guardar () o consultar () y no revela detalles internos de su implementación a otros módulos.
* Alto acoplamiento: Un módulo Carrito de compras que directamente accede a los detalles internos de un módulo Producto, como sus variables privadas, en lugar de utilizar métodos públicos.

**Relación entre cohesión y acoplamiento**

* **Alta cohesión** y **bajo acoplamiento** son características deseables en un sistema de software bien diseñado. Un sistema con alta cohesión asegura que cada módulo esté enfocado en una tarea específica, mientras que el bajo acoplamiento garantiza que los módulos sean lo más independientes posible entre sí, facilitando la modificación y mantenimiento del software.

**Pruebas**

**Pruebas Dinámicas: (Funcionales y no funcionales)**

Las pruebas dinámicas en el software son un conjunto de actividades que implican la ejecución real del programa o del código para evaluar su comportamiento y funcionamiento en tiempo de ejecución. Estas pruebas se centran en verificar que el software cumpla con los requisitos funcionales y de rendimiento establecidos y que funcione correctamente en diversas condiciones.

**Pruebas funcionales (Pruebas unitarias, integradas, end to end)**

**Pruebas Unitarias**

Las pruebas unitarias son aquellas que verifican de manera **individual** y **aislada** el comportamiento de las unidades más pequeñas del código, generalmente funciones, métodos o clases. La idea es asegurarse de que cada pieza de código funcione correctamente de manera independiente.

**Características:**

* **Aislamiento**: Las pruebas se realizan sobre unidades de código que se aíslan del resto del sistema para asegurar que los resultados no se vean afectados por otras dependencias.
* **Rápidas**: Son muy rápidas de ejecutar, ya que prueban una unidad pequeña de código.
* **Granularidad**: Cada prueba está enfocada en una única función o método.
* **Mocks y Stubs**: Es común utilizar **mocks** o **stubs** para simular el comportamiento de dependencias externas (bases de datos, APIs, etc.).

**Pruebas Integradas**

Las pruebas integradas verifican cómo funcionan entre sí **múltiples componentes o módulos** del sistema cuando se combinan o integran. Aquí se evalúa la interacción entre las distintas partes del sistema para asegurarse de que, cuando se combinen, sigan funcionando correctamente.

**Características:**

* **Prueban interacciones**: A diferencia de las pruebas unitarias, estas pruebas no se enfocan en probar módulos individuales, sino en cómo estos interactúan.
* **Medio alcance**: No prueban todo el sistema como las pruebas end-to-end, pero sí verifican la interacción entre componentes clave.
* **Dependencias reales**: Generalmente no se utilizan mocks o stubs, ya que se busca probar la integración real de los módulos (por ejemplo, interacción con una base de datos real).
* **Más lentas que las pruebas unitarias**: Ya que prueban múltiples partes del sistema, su ejecución puede tomar más tiempo.

**Ejemplo:**

Si la función de "calcular descuento" se integra con un sistema de pagos, una prueba integrada podría verificar que, al aplicar el descuento, el sistema de pagos calcula correctamente el monto total a cobrar.

**Pruebas End-to-End**

Las pruebas end-to-end verifican todo el sistema de **principio a fin**. Evalúan si la aplicación completa funciona correctamente al simular el comportamiento de un usuario o un flujo de trabajo completo. Esto incluye todas las capas del sistema, desde la interfaz de usuario hasta la base de datos.

**Características:**

* **Simulan el comportamiento del usuario**: Estas pruebas replican cómo interactuaría un usuario con el sistema (por ejemplo, navegando en la web, enviando formularios, etc.).
* **Cobertura amplia**: Cubren todos los componentes y flujos del sistema, asegurándose de que el sistema completo funcione correctamente en su conjunto.
* **Lentas**: Debido a la cobertura completa, suelen ser más lentas que las pruebas unitarias o integradas.
* **Difíciles de mantener**: Dado que prueban todo el sistema, cualquier pequeño cambio en la interfaz o flujo puede requerir la actualización de estas pruebas.

**Ejemplo:**

En una aplicación de comercio electrónico, una prueba end-to-end podría simular el proceso de un usuario que agrega productos al carrito, aplica un descuento, realiza el pago y recibe una confirmación.

**Pruebas No funcionales**

* Funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad, portabilidad, seguridad, compatibilidad, confiabilidad.

**Pruebas Estáticas:**

**¿Qué son las pruebas estáticas?**

* Las pruebas estáticas son un conjunto de técnicas utilizadas para evaluar el código sin necesidad de ejecutarlo.
* Son una parte fundamental del proceso de aseguramiento de la calidad del software.
* Se centran en encontrar errores en el código fuente antes de que se ejecute, lo que ayuda a reducir costos y mejorar la calidad del producto final.
* Las pruebas estáticas pueden identificar problemas como errores de sintaxis, malas prácticas de codificación, y posibles vulnerabilidades de seguridad.
* Son complementarias a las pruebas dinámicas, como las pruebas unitarias y las pruebas de integración, que se realizan durante la ejecución del software.

**Pruebas estáticas en el software**

* **Revisión de código:** En esta prueba, los miembros del equipo revisan el código fuente escrito por otros para buscar errores, inconsistencias y posibles mejoras. Las revisiones de código pueden ser formales o informales y son una de las formas más efectivas de identificar problemas en el código.
* **Análisis estático de código:** Se utilizan herramientas automatizadas para analizar el código fuente en busca de posibles problemas, como violaciones de estándares de codificación, uso incorrecto de variables, problemas de seguridad y más. Ejemplos de herramientas incluyen linters y analizadores estáticos.
* **Inspecciones:** Las inspecciones son revisiones formales y planificadas del diseño y la documentación del software. Los equipos de desarrollo se reúnen para revisar y discutir en detalle el diseño y la documentación del sistema, identificando posibles problemas y mejoras.
* **Pruebas de concepto:** En esta etapa, se evalúa la viabilidad de un concepto o diseño antes de comprometerse con un desarrollo completo. Puede implicar la creación de prototipos o simulaciones para validar ideas antes de invertir recursos significativos.
* **Análisis de impacto:** Se realiza un análisis de cómo un cambio propuesto en el código afectará al sistema en general y a otras partes del software. Esto ayuda a evaluar los riesgos y las implicaciones de los cambios antes de implementarlos.

**pre-commit**

* El precommit es una herramienta utilizada en el desarrollo de software para automatizar la ejecución de una serie de tareas, como análisis estático de código, formateo, pruebas unitarias, entre otros, antes de que los cambios sean confirmados en el repositorio de código.
* El precommit ayuda como validador estático al proporcionar una capa adicional de control de calidad antes de que los cambios sean incorporados al repositorio principal. Al ejecutar análisis estático de código durante el precommit, se pueden identificar y corregir problemas como errores de sintaxis, convenciones de estilo de codificación, posibles vulnerabilidades de seguridad y errores lógicos antes de que se integren en el código base.

**Introducción a DevOps**

**Qué es DevOps:** Filosofía de colaboración entre los equipos de desarrollo (Dev) y operaciones (Ops) para mejorar

la entrega de software.

**Objetivos de DevOps:** Aumentar la velocidad, mejorar la calidad y optimizar la eficiencia del desarrollo y

despliegue de software.

**Principios clave:**

* Automatización: Minimizar procesos manuales.
* Colaboración: Alinear equipos de desarrollo y operaciones.
* Medición: Monitorear y mejorar constantemente.

**Cultura DevOps:** Integración continua, entrega continua, automatización y retroalimentación constante.

**Cómo DevOps y CI/CD están relacionados:** CI/CD es un componente central para implementar DevOps.

**CI** y **CD** son prácticas de DevOps que ayudan a mejorar la velocidad, la calidad y la eficiencia en el desarrollo de software mediante la automatización. Estas siglas representan:

**CI (Integración Continua)**: **Continuous Integration** es el proceso de integrar cambios de código de varios desarrolladores en un proyecto común de forma frecuente y automática. El objetivo es detectar errores en una fase temprana. Cada vez que un desarrollador hace cambios en el código, este se integra en el repositorio central, y una serie de pruebas automáticas (como pruebas unitarias) validan el nuevo código. Esto asegura que el código funcione correctamente y no genere errores al combinarse con los cambios de otros desarrolladores.

**CD (Entrega Continua o Despliegue Continuo)**: Existen dos conceptos diferentes que suelen resumirse como **CD**, pero cada uno tiene una aplicación particular:

**Entrega Continua (Continuous Delivery)**: Implica que el código que pasa por todas las pruebas está siempre listo para desplegarse en producción. La entrega continua automatiza los procesos hasta el punto de que el equipo puede elegir cuándo hacer el despliegue, pero el código se mantiene listo para su lanzamiento en cualquier momento.

**Despliegue Continuo (Continuous Deployment)**: Lleva la entrega continua un paso más allá y automatiza el despliegue del código a producción. En este caso, cada cambio que pasa las pruebas se despliega automáticamente en el entorno de producción sin intervención humana. Esto permite realizar actualizaciones de forma continua y rápida para los usuarios.

ellos.

**SCRUM**

**SCRUM** es una metodología ágil de gestión de proyectos enfocada en el desarrollo de software que promueve la flexibilidad, la colaboración y la entrega continua de valor. En Scrum, los equipos trabajan en ciclos cortos llamados **sprints**, que suelen durar entre una y cuatro semanas. Al final de cada sprint, el equipo debe entregar una versión funcional del producto, lo que permite obtener retroalimentación constante y adaptar el proyecto de manera rápida.

**Principios de Scrum**

**Transparencia:** Todo el equipo debe tener una comprensión clara del estado del proyecto y de las tareas.

**Inspección:** El equipo revisa regularmente el trabajo realizado para detectar y corregir problemas.

**Adaptación:** Basado en la retroalimentación, el equipo ajusta su trabajo para mejorar la entrega en el próximo sprint.

**Patrones de diseño**

Existen muchos patrones de diseño en ingeniería de software, pero los tres tipos principales en los que se agrupan son: **creacionales**, **estructurales** y **comportamentales**. Estos patrones ofrecen soluciones probadas para problemas recurrentes en el desarrollo de software orientado a objetos. A continuación, se describen estos tres tipos de patrones y algunos ejemplos dentro de cada grupo.

**1. Patrones Creacionales**

Estos patrones se centran en la creación de objetos, ayudando a que la creación sea más flexible y reutilizable. El objetivo es separar el proceso de creación de objetos de su uso, de modo que el código no dependa de las clases concretas.

**Ejemplos**:

* **Factory Method**: Proporciona una interfaz para crear objetos, pero permite a las subclases decidir qué clase instanciar.
* **Abstract Factory**: Crea familias de objetos relacionados sin especificar sus clases concretas.
* **Singleton**: Asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.
* **Builder**: Separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción cree diferentes representaciones.

**Caso de uso**: Cuando necesitas controlar la creación de objetos de manera flexible o cuando el proceso de creación de objetos es complejo.

**2. Patrones Estructurales**

Los patrones estructurales se centran en la forma en que se componen las clases y los objetos para formar estructuras más grandes. Estos patrones ayudan a garantizar que el sistema esté estructurado de manera eficiente y manejable.

**Ejemplos**:

* **Adapter**: Permite que dos clases incompatibles trabajen juntas proporcionando una interfaz que las une.
* **Decorator**: Añade comportamiento a los objetos de manera dinámica sin modificar su clase.
* **Facade**: Proporciona una interfaz simplificada a un conjunto de clases más complejas, haciendo el sistema más fácil de usar.
* **Proxy**: Proporciona un objeto sustituto que controla el acceso a otro objeto, pudiendo gestionar diferentes tipos de control, como la creación diferida o la autenticación.

**Caso de uso**: Cuando necesitas cambiar la interfaz de clases existentes, añadir responsabilidades de forma dinámica o controlar el acceso a objetos.

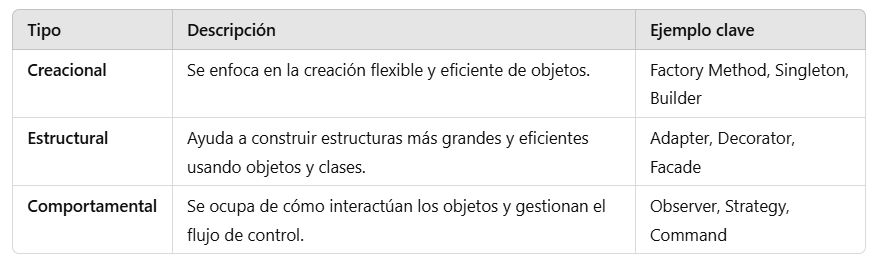
**3. Patrones Comportamentales**

Los patrones comportamentales se centran en cómo interactúan los objetos y cómo se distribuyen las responsabilidades entre ellos. Estos patrones facilitan la comunicación y cooperación entre los objetos de manera flexible y eficiente.

**Ejemplos**:

* **Observer**: Define una relación de dependencia entre objetos de manera que, cuando uno cambia de estado, los dependientes son notificados automáticamente.
* **Strategy**: Permite seleccionar un algoritmo en tiempo de ejecución al encapsular cada uno en una clase separada.
* **Command**: Encapsula una solicitud como un objeto, permitiendo parametrizar a los clientes con diferentes solicitudes, colas o registros.
* **Mediator**: Define un objeto que encapsula cómo interactúan un conjunto de objetos, promoviendo un acoplamiento débil entre ellos.

**Caso de uso**: Cuando es importante definir la comunicación entre objetos sin crear dependencias rígidas o cuando necesitas que varios objetos respondan a cambios en otros objetos.



**Modelos tradicionales en el ciclo de vida del desarrollo de software**

**Modelo en cascada:** Su proceso secuencial, cada fase debe completarse antes de que comience la siguiente, lo que da la apariencia de una cascada de pasos.

**Características del Modelo en Cascada**

* **Secuencial y Rígido**: Las fases siguen una secuencia específica, lo que significa que no se pasa a la siguiente fase hasta que se haya completado la anterior.
* **Documentación Completa**: Cada fase suele tener una documentación exhaustiva que sirve de referencia y facilita la transición entre fases.
* **Ideal para Proyectos con Requisitos Claros**: Es útil cuando los requisitos son bien conocidos desde el inicio y no se espera que cambien.

**Modelo en espiral:** Es un modelo de desarrollo de software que combina características del modelo **en cascada** y del modelo **iterativo**, con un enfoque en la gestión de riesgos. Se usa principalmente en proyectos grandes, complejos o de alto riesgo. El modelo en espiral se basa en el concepto de **repetición cíclica** de las fases del desarrollo de software, permitiendo abordar problemas y mejorar el sistema en cada ciclo, o "espiral".

**Características Clave del Modelo Espiral**

* **Enfoque en la Gestión de Riesgos**: La identificación y mitigación de riesgos es un aspecto central, lo que lo hace adecuado para proyectos con alta incertidumbre o riesgo.
* **Iterativo e Incremental**: El desarrollo se realiza en iteraciones o ciclos, lo que permite refinar el sistema progresivamente.
* **Flexibilidad y Adaptabilidad**: Se puede adaptar a los cambios en los requisitos y permite ajustar los objetivos según las necesidades del proyecto.
* **Prototipos Tempranos**: Los prototipos ayudan a reducir los riesgos al brindar al cliente una vista preliminar del sistema en desarrollo.

**Modelo en V:**

El Modelo V es una extensión del modelo en cascada que destaca la relación entre desarrollo y pruebas.

Cada fase de desarrollo se asocia con una fase de validación o prueba.

La V se expande a medida que avanza el proceso, garantizando pruebas en cada etapa para cumplir con los requisitos establecidos.

**Características del Modelo en V**

* **Estructura Rígida**: Similar al modelo en cascada, el modelo en V es secuencial y rígido, donde cada fase tiene una prueba asociada que verifica su correcto desarrollo.
* **Enfoque en Verificación y Validación**: Cada etapa de desarrollo tiene una fase de prueba específica para garantizar la calidad y que se cumplan los requisitos.
* **Alta Dependencia de Documentación**: La documentación es importante, ya que cada fase debe completarse de manera exhaustiva antes de pasar a la siguiente.

**Historias de usuario**

**¿Qué es una historia de usurario?**

Una historia de usuario es una descripción sencilla de una función de software desde la perspectiva del usuario final o cliente.

* **Como** cliente de un restaurante, **quiero** poder ver el menú en línea, **para** decidir mi pedido antes de llegar.

**Una historia de usuario debe de incluir:**

* Titulo
* Descripción
* Criterios de validación

**Requisitos Funcionales:** Están enfocado en lo que el sistema debe hacer.

* El sistema debe permitir a los usuarios registrarse con un correo electrónico y una contraseña.
* El sistema debe de enviar un correo de notificación al usuario después de realizar la compra.

**Requisitos No funcionales:** Enfocados en la calidad del software.

* Funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad, portabilidad, seguridad, compatibilidad, confiabilidad.

**Diagramas de arquitectura**

**¿Qué son los diagramas de arquitectura?**

Un diagrama de arquitectura es una representación visual que traza la implementación física de los componentes de un sistema de software.

Muestra la estructura general del sistema de software y las asociaciones, limitaciones y límites entre cada elemento.

**Tipo de diagramas de arquitectura:**

* Diagrama de arquitectura de aplicación.
* Diagrama de arquitectura de despliegue.
* Diagrama de arquitectura de Secuencias (Casos de uso).
* Modelo C4 (Contexto, contenedores, componentes).

**Arquitectura de software**

**¿Qué es la arquitectura de software?**

* Estructura organizativa fundamental de un sistema de software.
* Incluye componentes principales, sus relaciones e interacciones.
* Comparable a un plano maestro de un edificio.
* Define partes clave y sus relaciones para formar una solución coherente y funcional.
* Enfocada en aspectos técnicos y en cumplir requisitos funcionales y no funcionales.

**Arquitectura Hexagonal**

También conocida como **Arquitectura de Puertos y Adaptadores**. Su objetivo es hacer que la aplicación sea más flexible, modular y fácil de probar, desacoplando la lógica de negocio de los detalles externos como las interfaces de usuario, las bases de datos o los servicios externos.

**Principios clave:**

* **Separación de capas**: La lógica de negocio de la aplicación reside en el centro del hexágono (núcleo), mientras que los detalles de implementación externos (como la base de datos, la interfaz de usuario o las API) están en los bordes.
* **Independencia de los componentes externos**: Los componentes como bases de datos, APIs o servicios de terceros se comunican con la lógica de negocio a través de **puertos** y **adaptadores**, lo que permite intercambiar o modificar estos elementos sin afectar el núcleo de la aplicación.
* **Facilidad para pruebas**: Al estar la lógica de negocio desacoplada de las dependencias externas, es más fácil de probar. Se pueden reemplazar fácilmente los adaptadores externos por stubs o mocks en los tests.

**Estructura básica:**

* **Core (Núcleo)**: Contiene la lógica de negocio y las entidades. Es independiente de cualquier tecnología externa.
* **Puertos (Ports)**: Interfaces que define el núcleo para interactuar con los componentes externos.
* **Adaptadores (Adapters)**: Implementan las interfaces definidas en los puertos, permitiendo que las tecnologías externas (bases de datos, APIs, interfaces de usuario, etc.) interactúen con el núcleo.

**Ejemplo:**

Imagina una aplicación de gestión de pedidos. El núcleo del hexágono contiene la lógica de procesamiento de pedidos. Un puerto define una interfaz que la aplicación necesita para enviar correos electrónicos de confirmación. Un adaptador implementa este puerto usando una API externa de email, pero si mañana cambias a otro proveedor de email, solo necesitas ajustar el adaptador sin tocar el núcleo.

**Arquitectura Orientada a Eventos (EDA)**

La **arquitectura orientada a eventos** se basa en la producción, detección y reacción a eventos dentro de un sistema. Un **evento** es un cambio de estado o una acción ocurrida en la aplicación que puede desencadenar respuestas de otros componentes del sistema. EDA es utilizada para crear sistemas distribuidos, altamente escalables y desacoplados.

**Principios clave:**

* **Productores y consumidores**: Los productores son componentes que generan eventos (por ejemplo, cuando un usuario realiza una acción), mientras que los consumidores son componentes que reaccionan a estos eventos (por ejemplo, procesando la acción y actualizando una base de datos).
* **Desacoplamiento**: Los componentes no dependen directamente entre sí, ya que se comunican a través de eventos. Esto reduce el acoplamiento entre los distintos servicios.
* **Asincronía**: Los eventos permiten que las operaciones sean asincrónicas. Un productor no espera a que los consumidores procesen el evento, lo que mejora el rendimiento y la capacidad de respuesta del sistema.

**Patrones comunes en EDA:**

* **Cola de mensajes (Message Queue)**: Los eventos son colocados en una cola y los consumidores los procesan uno a uno. Esto es útil para gestionar eventos en orden y garantizar la entrega.
* **Publicación/Suscripción (Pub/Sub)**: Los productores publican eventos y los consumidores se suscriben para recibir solo los eventos de interés. Este patrón es común en sistemas distribuidos y microservicios.
* **Event Sourcing**: En lugar de almacenar el estado final de una entidad, se almacenan todos los eventos que llevaron a ese estado. Esto permite reconstruir el estado en cualquier momento basado en los eventos.

**Ejemplo:**

En un sistema de e-commerce, cuando un cliente realiza un pedido, se genera un evento "Pedido Creado". Este evento puede ser consumido por diferentes servicios: uno para enviar un email de confirmación, otro para actualizar el inventario, y otro para generar la factura, todos trabajando de forma independiente y asincrónica.

**arquitectura monolítica**

La **arquitectura monolítica** es un enfoque tradicional de desarrollo de software donde toda la funcionalidad de una aplicación se encuentra en una única base de código o aplicación. En este tipo de arquitectura, todos los componentes y funcionalidades de la aplicación (como la lógica de negocio, la interfaz de usuario, la gestión de datos y las integraciones con otros sistemas) están estrechamente integrados en un único programa o servicio.

**Definición de Cobertura de Código y Cobertura de Pruebas**

* Code Coverage: Es una métrica que evalúa qué parte del código fuente ha sido ejecutada al menos una vez durante la ejecución de pruebas. Incluye la cobertura de sentencias, ramas y caminos.
* Test Coverage: Se refiere a la medida en que un conjunto de casos de prueba cubre el código del programa. Incluye la cobertura funcional, estructural y de rutas críticas.

**Tipos de Cobertura**

* Statement Coverage: Evalúa si cada sentencia del código ha sido ejecutada al menos una vez durante las pruebas. No garantiza que todas las ramas hayan sido evaluadas.
* Branch Coverage: Examina si todas las ramas de control del código, como los condicionales y los bucles, han sido ejecutadas al menos una vez.
* Path Coverage: Analiza si todas las posibles rutas de ejecución del código, incluyendo combinaciones de ramas, han sido cubiertas por las pruebas.