Calidad de SW y Modularidad:

Se define al "usuario" como aquel que utiliza el sistema y también aquel que compra el sistema o contrata su desarrollo. Factores de calidad: Correctitud, robustez, extendibilidad, reusabilidad, eficiencia, portabilidad, facilidad de uso, funcionalidad, modularidad, cohesión, acoplamiento, legibilidad. (definiciones)

Modularidad: Un módulo se refiere simplemente a los componentes de software creados mediante la división del software. El software se divide en varios componentes que trabajan en conjunto para formar un único elemento funcional, aunque a veces pueden funcionar como una función completa incluso sin estar conectados entre sí. El principio básico de la modularidad es que "los sistemas deben construirse a partir de componentes cohesivos y débilmente acoplados (módulos)". Cada parte se focaliza en un aspecto del sistema, son intercambiables e independientes. El buen diseño facilita la reusabilidad.

Criterios de Modularidad: definen las metas que se buscan al diseñar un sistema con buena modularidad:

- ➢ Descomponibilidad Modular: La capacidad de descomponer un problema grande en subproblemas más pequeños e independientes.
- ➢ Composición Modular: La capacidad de combinar módulos existentes para crear nuevos sistemas o funcionalidades.
- ➢ Comprensión Modular: Un módulo debe ser comprensible y analizable por sí mismo, con una mínima dependencia de otros módulos.
- ➢ Continuidad Modular: Favorece que un cambio pequeño en la especificación impacte en uno o pocos módulos.
- ➢ Protección Modular: Un error o condición anormal en un módulo debe permanecer localizado y no propagar el fallo a otros módulos.

Reglas de Modularidad: pautas de diseño que ayudan a alcanzar los criterios de modularidad:

- ➢ Pocas Interfaces: Cada módulo debe interactuar con la menor cantidad posible de otros módulos. (Favorece el acoplamiento débil).
- ➢ Interfaces Pequeñas: Si dos módulos interactúan, deben intercambiar la menor cantidad posible de información, limitándose a una interfaz mínima.
- ➢ Interfaz Explícita: Cuando los módulos interactúan, su comunicación debe ser explícita y visible en su definición. No debe haber efectos laterales ocultos.
- ➢ Ocultamiento de la Información: La implementación interna de un módulo debe ser privada y sólo debe exponer su interfaz pública.
- ➢ Mapeo directo: Intentar que el diseño del sistema refleje o "mapee" lo más fielmente posible el modelo conceptual del dominio de la realidad. Si un problema real tiene conceptos de

Principios de Diseño de Modularidad: Estos son los pilares fundamentales del diseño orientado a objetos que impactan directamente la calidad de los módulos:

- ➢ Acoplamiento Débil: Los módulos deben depender unos de otros lo menos posible. Esto se logra usando interfaces y composición (en lugar de herencia).

➢ Alta Cohesión: Los elementos dentro de un módulo deben estar altamente relacionados y trabajar juntos hacia un solo propósito bien definido. Esto mejora la comprensibilidad y la mantenibilidad.

SOLID: (principios de modularidad)

Conjunto de directrices que procuran facilitar la creación de sw de calidad. Representan fundamentos relevantes de la arquitectura y desarrollo de sw. Su objetivo es favorecer la flexibilidad, reusabilidad y extensibilidad.

- Principio de Responsabilidad Única (SRP): Una clase debe tener solo una razón para cambiar, una única responsabilidad que puede motivar este cambio.
- Principio Abierto Cerrado (OCP): las entidades de sw deben estar abiertas a extensiones, pero cerradas a modificaciones. Debe ser posible extender al comportamiento de las entidades, y al mismo tiempo se debe mantener intacto el comportamiento original. Toda entidad manejada desde una abstracción puede cerrarse a modificaciones, y puede extenderse creando nuevas entidades derivadas.
- Principio de Sustitución de Liskov (LSP): Las clases derivadas deben poder sustituir cualquier aparición de sus superclases sin alterar el comportamiento del programa.
- Principio de Segregación de Interfaces (ISP): Los clientes no deben verse obligados a depender de interfaces que no utilizan, deben permanecer los más pequeñas posibles.
- Principio de Inversión de Dependencia (DIP): Los módulos, si dependen de otros, deben hacerlos de aquellos del mayor nivel de abstracción posible. Fomenta el desacoplamiento y la flexibilidad, especialmente entre módulos de alto y bajo nivel. Las abstracciones no deben depender de detalles, sino que los detalles deben depender de las abstracciones.

Diseño de un sistema: se modela para proveer una estructura para la solución del problema, proveer las abstracciones necesarias para lidiar con la complejidad, reducir los costos de desarrollo y minimizar el riesgo de cometer errores. Modelar es simplificar la realidad. Modelamos para comprender lo que construimos

Programación Orientada a Objetos y Diseño Modular:

El POO favorece la modularidad de un sistema de software de manera fundamental, ya que sus principios básicos están diseñados para implementar los criterios de modularidad, como la Cohesión Fuerte y el Acoplamiento Débil. Los principales objetivos son: Confiabilidad, Extensibilidad y Reusabilidad. Define tres conceptos claves: Clase, Objeto y Mensaje. Y tres mecanismos claves: Encapsulado, Polimorfismo y Herencia

Las instancias de una clase son los objetos que son instancia de algún descendiente de la clase. Las instancias propias son las instancias de la misma clase. Todas las instancias de una clase pueden reemplazar a una instancia propia, dado que al menos cumplen la misma funcionalidad. Por esto un objeto puede ser de varios tipos de datos. Una clase base se dice es una clase frágil cuando una modificación aparentemente segura en la clase base implica un incorrecto funcionamiento en la clase derivada

Un lenguaje orientado a objetos es estáticamente tipado si está equipado con un conjunto de reglas de consistencia, controlada por los compiladores, cuya observancia por el texto del software garantiza que la ejecución del software no incurrirá en una violación de tipos.

Toda entidad debe tener un tipo declarado antes de ser utilizada. El tipo dinámico de una entidad x en un determinado momento de ejecución es la clase de la que es instancia directa el objeto asociado a x en ese momento. El tipo estático de una entidad x es el tipo usado para declarar.

Una asociación polimórfica ocurre cuando a una referencia de una clase se le asocia una referencia a una instancia no propia. Una entidad como p, que aparece en una asignación polimórfica de este tipo es denominada entidad polimórfica.

El concepto de Clase puede definirse como un tipo de dato abstracto posiblemente equipado con una implementación total o parcial. Una clase es un módulo (unidad de composición y descomposición de sw) y un tipo de datos (descripción estática de un conjunto de valores y de las operaciones que se pueden efectuar sobre ellas). Define cómo el objeto es implementado: el estado interno y las implementaciones concretas de sus operaciones. El tipo sólo se refiere a la interfaz, es decir, al conjunto de pedidos que ese objeto es capaz de responder.

La herencia de interfaz describe cuándo un objeto puede ser usado en lugar de otro, porque al fin y al cabo, responden a los mismos pedidos. Provee un menor acoplamiento. La herencia de clase define la implementación de un objeto en términos de la implementación de otro objeto, es, en parte, un mecanismo para reutilizar código.

Patrones de Diseño:

Un patrón describe un problema recurrente de clases y objetos comunicantes, y ofrecen soluciones que ayudan a crear diseños más flexibles, elegantes y reutilizables. Es una solución reutilizable y probada a un problema común en el diseño de software. Tiene ciertos elementos: Nombre, problema (cuando aplicarlo), solución (cómo aplicarlo), y consecuencias. Se dividen en un scope de clase u objeto, y pueden tener un propósito creacional, estructural o de comportamiento.

- Creacionales: abstraen el proceso de instanciación. Procuran independizar el sistema de cómo sus objetos son creados, compuestos y representados. Indican soluciones de diseño para encapsular el conocimiento acerca de las clases que el sistema usa y ocultar cómo se crean instancias de estas clases.
- Estructurales: se refieren a cómo las clases y los objetos son organizados para confirmar estructuras más grandes. Pueden ser patrones de clases, basados en la utilización de herencia, o patrones de objetos, basados en la técnica de composición. Composite es un ejemplo. Adapter es un patrón estructural que puede clasificarse como un patrón del scope Class y Object.
- De Comportamiento: se centran en los algoritmos y la asignación de responsabilidades entre los objetos. Son tanto de clases, que utilizan herencia, y objetos, que utilizan composición, como de comunicación entre ellos. Observer es uno, Strategy es otro que puede clasificarse como un patrón de scope Object.

Modelos de 3 Capas:

El modelo de 3 capas, también conocido como arquitectura de tres capas, es un patrón de diseño utilizado en el desarrollo de software que propone estructurar y organizar todo el sistema a partir de 3 niveles bien distinguidos. Mejora la organización de un proyecto,

permite la separación de responsabilidades, facilita el mantenimiento, permite la reutilización, favorece la escalabilidad, y simplifica el testing. Las capas permiten pensar soluciones organizadas y separando las responsabilidades.

- Vista: partes que se encargan de la presentación y visualización de la información e interacción con el usuario. Se implementan los elementos visuales, y se maneja la interacción traduciendo acciones en eventos que se envían a la lógica de negocio para su procesamiento.
- Lógica: parte que se encarga de procesar la información y aplicar las reglas de negocio. Su función principal es gestionar la lógica de la aplicación. Interpreta las solicitudes y ejecuta las operaciones necesarias. Actúa como intermediaria entre vista y datos.
- Datos: se encargan de la gestión y persistencia de la información. Proporciona acceso a las diferentes fuentes o bases de datos. Se encarga de manejar las operaciones relacionadas con el almacenamiento, recuperación, actualización y eliminación de datos. Abstrae los detalles específicos del almacenamiento y permite que la lógica interactúe con los datos sin conocer la implementación subyacente.

Facilita la separación de responsabilidades, mantenimiento y la escalabilidad. El modelo de arquitectura de 3 capas está diseñado específicamente para lograr una fuerte separación de preocupaciones. Los cambios en una capa pueden realizarse sin afectar directamente a la lógica central de la aplicación o la forma en que se almacenan los datos. Esto hace que la aplicación sea más fácil de mantener porque los bugs o las actualizaciones están aislados. La separación permite que cada capa sea escalada de forma independiente. Si bien la reutilización de componentes y colaboración entre equipos son beneficios derivados importantes, la separación de responsabilidades es el principio fundamental y el motor directo que permite la mejora en el mantenimiento y la escalabilidad.

Concurrencia:

Concurrencia es ejecutar simultáneamente varios programas, mientras que paralelismo es la ejecución simultánea de varios procesadores. Thread-safety implica que el código es seguro de ser utilizado por diferentes hilos de ejecución, manteniendo la consistencia pretendida. La sincronización es necesaria para los datos compartidos y asegurar la consistencia. La concurrencia requiere especial atención en los aspectos de seguridad, ciclo de vida, no determinismos y tiempo de ejecución.

Java permite programación multithread. Una operación de un objeto puede ejecutarse en threads diferentes, bajo memoria compartida. Existen dos formas de obtener objetos con código concurrente: Heredar la clase Thread, que sería la mejor abstracción pera tendrá todos los métodos; o implementar la interfaz runnable pero puede ser menos simple. Ambas tienen los métodos run() y start().

Cada objeto tiene asociado un lock, que permite restringirla concurrencia en porciones de código: sólo un objeto puede acceder al lock al mismo tiempo, los demás deben esperar. Las operaciones declaradas como synchronized impiden la ejecución intercalada entre varios threads, previenen la interferencia entre threads y ayudan a controlar la consistencia.

Un deadlock ocurre cuando dos o más hilos están bloqueados indefinidamente, esperando recursos que los otros hilos bloqueados tienen en su poder. Para que ocurra un deadlock,

deben cumplirse cuatro condiciones de Coffman. Prevenir cualquiera de ellas evita el deadlock. Orden estricto de Locks: Al obligar a todos los hilos a adquirir los locks en un orden predefinido global, se rompe la posibilidad de que se forme un ciclo de espera. Si el Hilo 1 necesita A y luego B, y el Hilo 2 necesita B y luego A, esto puede llevar a un deadlock. Si ambos están obligados a adquirir siempre primero A y luego B, el primero que adquiera A obligará al otro a esperar, pero el deadlock no ocurrirá.

Herencia Múltiple:

Una clase puede ser una combinación de otras clases. En este caso dos o más clases son generalizaciones parciales de conceptos que admiten más de una vista. Pero esto puede generar una colisión de nombres o ambigüedad de nombres cuando dos servicios son *efectivos*, invalidando la clase. Otro problema que genera es la herencia repetida o problema diamante, ocurre cuando una clase es descendiente de otra en más de una forma.

Refactoring:

Proceso de cambiar un sistema de sw disciplinadamente de forma tal que no altere el comportamiento externo, mejorando su estructura interna. Los Bad Smells en código son indicios de posibles problemas, señales de que se necesita refactorizar.

El objetivo principal es mejorar la calidad interna del código para que sea más fácil de entender, mantener y modificar en el futuro. Busca aumentar la legibilidad y mantenibilidad, disminuir la complejidad y errores, y mejorar el diseño. Es un ciclo continuo y disciplinado, diseñado para minimizar el riesgo de introducir errores:

1. Identificar el código a refactorizar (Code Smell): Identificar una parte del código que necesita ser mejorada. Esto podría ser un método demasiado largo, una clase con baja cohesión, una duplicación de código o un mal nombre de variable.
2. Crear pruebas unitarias: Asegurarse de que los cambios no introduzcan nuevos errores. Se busca verificar que el comportamiento externo del código no cambie después de la refactorización.
3. Realizar cambios pequeños y graduales: Aplicar las técnicas de refactorización de forma incremental. El cambio debe ser mínimo para facilitar la identificación de fallos.
4. Verificar los cambios: se ejecutan las pruebas unitarias en busca de posibles errores.
5. Repetir: Una vez verificado, se vuelve al paso 1 hasta alcanzar la calidad deseada.

Diseño por Contrato:

Para que un programa sea correcto se necesita saber la descripción precisa de lo que se supone que debe hacer el programa, la especificación. Diseño por contrato entiende las relaciones entre una clase y sus clientes como un acuerdo formal expresando obligaciones y derechos a cada uno de los participantes. Las aserciones son mecanismos para definir las especificaciones del sistema.

Las aserciones son expresiones que involucran algunas entidades del sw y establecen unas propiedades que éstas entidades deben cumplir en un momento de la ejecución. Las fórmulas de correctitud son expresiones de la forma {P} A {Q}. La precondición establece las propiedades que se tienen que cumplir cada vez que se llame a la operación, y la postcondición establece las propiedades que debe garantizar la operación cuando retorne.

Pueden tener distintos niveles de fortaleza, la fórmula P1 es más fuerte que P2 si P1 implica P2 y no son iguales. En herencia, para no interferir en el acuerdo de la clase base, si hay una redefinición se debe incluir un requerimiento igual o más débil que el original.

Un error es una decisión equivocada hecha durante el desarrollo, un defecto es una característica del sw que puede provocar una desviación de sus objetivos, y una falla es el hecho de que el sw se desvíe de sus objetivos. Las aserciones pretenden evitar los defectos y favorecer la confiabilidad. Esto se relaciona estrechamente con la Sustitución de Liskov (LSP): la subclase no puede violar la condición de integridad definida por la superclase, reforzando la confiabilidad y se asegura que el subtipo sea un sustituto seguro.

Una invariante de clase es una aserción que expresa restricciones generales que se aplican a toda la clase. Expresa una propiedad que debe cumplirse en todo estado estable de una instancia de clase, inmediatamente después de ejecutarse el constructor y, antes y después de responder a cualquier comando. Es correcta si: todo procedimiento de creación de la clase y toda rutina exportada concluye en un estado que la satisface (cumpliendose P y Q).

Java provee la sentencia assert que permite testear condiciones de ejecución. Aunque no cumple completamente con el diseño por contrato, no están diseñadas para establecer contratos. Su propósito es ayudar al desarrollador a detectar bugs. No distinguen responsabilidades que deben ser cumplidas, solo verifican condiciones locales. Los assert pueden desactivarse en entornos de producción.

Excepciones:

Panico Organizado: Es una estrategia de manejo de excepciones y errores de manera controlada y predecible, incluso en situaciones críticas. Es una de las respuestas legítimas a una excepción dentro de una política disciplinada de manejo de errores, especialmente en el contexto del Diseño por Contrato. Es la acción que toma un manejador de excepciones cuando no es posible una recuperación exitosa del fallo. En este escenario, el objetivo no es reintentar la operación o capturar y continuar, sino fallar de manera controlada y propagar la falla

Se implementa en Java cuando un catch block determina que el error es irrecuperable y, en lugar de manejarlo, lanza una nueva excepción o la re-lanza para que se propague, forzando la terminación del proceso actual y notificando al llamador.

- try-catch: Bloques para manejar excepciones. El bloque try contiene el código que puede generar una excepción, y el bloque catch maneja la excepción si se produce.
- throws: Se utiliza para declarar que un método puede generar una excepción, lo que obliga al llamar a manejarla o declararla.
- finally: Un bloque que se ejecuta siempre después de un bloque try, independientemente de si ocurrió una excepción o no. Es útil para liberar recursos o realizar tareas de limpieza.

Las excepciones chequeadas son las que el compilador de Java obliga a manejar con un bloque try-catch o a declarar explícitamente en la firma. Representan condiciones externas o de entorno que son recuperables, pero que el programa debe prever. Mientras que las unchecked son un subconjunto que el compilador no obliga a manejar o declarar. Representan errores de programación o fallos graves del sistema. Se asume que estos

errores deberían haberse evitado mediante una lógica de programación correcta. El manejo es opcional. Si no se capturan, se propagan hasta el hilo principal y generalmente detienen la ejecución del programa.

Preguntas de Promoción:

1. Analice y explique cuáles son los principales inconvenientes y desventajas al utilizar instanceOf.

El operador instanceof en Java se utiliza para verificar si un objeto es una instancia de una clase particular, una subclase de esa clase, o si implementa una interfaz determinada. Su uso excesivo o inapropiado genera acoplamiento y atenta contra el principio de Abierto/Cerrado (OCP) de SOLID, considerado el principal inconveniente.

El uso frecuente de instanceof crea un fuerte acoplamiento entre clases, lo que dificulta la modificación y el mantenimiento del código. Esto hace que agregar nuevas clases sea más complicado, ya que obliga a cambiar el código en varios lugares para que funcione. El OCP establece que las entidades de software deben estar abiertas a la extensión pero cerradas a la modificación. Si se usa instanceOf, cada vez que se añade una nueva subclase, el código que contiene el instanceof debe ser modificado para incluir una nueva rama, rompiendo el OCP.

El uso de instanceof anula el beneficio principal de la POO: el polimorfismo. Este permite tratar objetos de diferentes tipos como si fueran del mismo tipo, siempre y cuando compartan una interfaz común. Al utilizar instanceof, estamos verificando el tipo exacto de un objeto en tiempo de ejecución, lo que va en contra de la idea de polimorfismo. En lugar de usar instanceof para decidir qué método llamar, se debe definir el método en la interfaz común y dejar que las subclases proporcionen su propia implementación. Se recomienda el uso del patrón Visitor y las colecciones genéricas, que permiten tratar los objetos de manera más abstracta y flexible.

2. Explique por qué un objeto puede ser de varios tipos de datos.

Un objeto puede ser de varios tipos de datos debido a los pilares fundamentales de la Programación Orientada a Objetos: Herencia y Polimorfismo. Cuando una clase hereda de otra, se establece una relación jerárquica donde la subclase es también considerada del tipo de la superclase. Si la clase Perro hereda de la clase Animal, un objeto de tipo Perro es un Perro, pero por herencia, también es un Animal.

El polimorfismo es la capacidad de una variable de referencia para tomar la forma de diferentes tipos de objetos. Una variable declarada de tipo Animal puede referenciar un objeto de tipo Gato o un objeto de tipo Perro. Aunque el objeto real en memoria sea un Gato (su tipo real), la variable lo trata como un Animal (su tipo declarado). El tipo de referencia (la variable) puede ser más general (una superclase o una interfaz), mientras que el tipo real del objeto en memoria es la clase concreta con la que se creó.

3. ¿Qué aspectos de la Programación Orientada a Objetos colaboran con la reusabilidad?

La reusabilidad es la capacidad de utilizar componentes de software existentes para construir nuevas aplicaciones o módulos, minimizando la duplicación de código. Varios pilares y conceptos de la POO están diseñados para maximizarla:

- La Herencia es el mecanismo más directo para la reusabilidad de implementación. Permite que una subclase herede métodos y atributos de una superclase. El código escrito en la clase padre se reutiliza directamente en la clase hija sin necesidad de reescribirlo.
- La Abstracción se centra en mostrar solo los detalles relevantes y ocultar la implementación compleja. Las Interfaces y las Clases Abstractas definen un contrato común sin la implementación. Esto fomenta el desarrollo de componentes genéricos que pueden interactuar con cualquier objeto que cumpla el contrato, promoviendo la reutilización de diseño.
- El Encapsulamiento es el empaquetamiento de datos y código en una sola unidad, y la restricción del acceso directo al estado interno. Se utiliza la ocultación de información. Al ocultar los detalles de implementación, los módulos se vuelven independientes (bajo acoplamiento). Si el código interno de un módulo cambia, su interfaz pública no se ve afectada, lo que permite que otros módulos sigan reutilizándolo sin fallar.
- El polimorfismo permite tratar diferentes objetos de manera uniforme a través de una interfaz común. El código opera sobre una interfaz en lugar de una implementación concreta. Esto permite la Inyección de Dependencias. Se puede reutilizar el mismo código cliente, simplemente inyectándole una implementación diferente de la interfaz en tiempo de ejecución.

4. Explique cuál es el objetivo del uso de métricas de software. Relacionar con medir factores de calidad. Describa al menos tres métricas para programación orientada a objetos que usted considere importantes. Explique por qué. Describa al menos una métrica para programación orientada a objetos que usted considere poco útil.

El objetivo del uso de métricas de software es proporcionar datos cuantificables y objetivos que permitan medir, monitorear y controlar tanto el proceso de desarrollo como el producto final. Buscan evaluar la calidad del software. Las métricas de software nos permiten cuantificar atributos de calidad como: para el factor *mantenibilidad* se mide el acoplamiento y la cohesión, que tan fácil es modificar y reparar el código a lo largo del tiempo. Las métricas permiten identificar áreas problemáticas del código, estimar el esfuerzo de mantenimiento o la probabilidad de fallos futuros, y evaluar la efectividad de diferentes técnicas de desarrollo, equipos o versiones de software. Métricas importantes:

- Número de métodos por clase: Un alto valor de métodos en una clase puede indicar una clase con demasiadas responsabilidades, lo que dificulta su comprensión y mantenimiento.
- Acoplamiento entre clases: Mide la dependencia de una clase con otras. Si se encuentra fuertemente acoplado esto dificulta los cambios y la reutilización de código.
- Profundidad de herencia: Indica la distancia de una clase a la raíz de la jerarquía de herencia. Una alta profundidad de clases puede indicar una jerarquía compleja y difícil de mantener.
- Una métrica poco útil: Número de líneas de código, no considera factores importantes como la complejidad algorítmica, la legibilidad del código o la eficiencia.

5. Mencione alguna característica de calidad del software que pueda verse afectada por el refactoring. Explicar por qué.

Funcionalidad (Fiabilidad/Confiabilidad): si el software hace lo que se supone que debe hacer. Es la capacidad del software de mantener un nivel de rendimiento especificado durante un período de tiempo dado, es decir, funcionar sin fallos. El refactoring es inherentemente una actividad de alto riesgo porque implica modificar la lógica del código existente. Cualquier modificación al código introduce la posibilidad de un error humano, lo que resulta en un comportamiento diferente al esperado (regresión). Además, si la porción de código que se está refactorizando no está cubierta por pruebas unitarias robustas, o si las pruebas existentes son deficientes, el desarrollador no tendrá una red de seguridad. El riesgo de comprometer temporal o permanentemente la funcionalidad o la fiabilidad es el costo directo que se debe mitigar a través de un proceso de testeo riguroso.

Mantenibilidad: al mejorar la estructura y la organización del código, se facilita la comprensión y modificación posterior. Simplificar el código hace que sea más fácil realizar cambios sin introducir nuevos errores. Al agrupar funcionalidades relacionadas, se reduce el acoplamiento entre diferentes partes del sistema. Pero, si las refactorizaciones no se realizan de forma cuidadosa y planificada, pueden generar código más complejo, menos legible y con mayor propensión a errores, lo que dificulta futuras modificaciones y aumenta el costo de mantenimiento. Realizar cambios superficiales o sin un plan claro puede llevar a resultados contraproducentes, como una mayor complejidad del código y una disminución en su calidad a largo plazo

6. Explique claramente el patrón de diseño Strategy.

El Patrón de Estrategia (Strategy Pattern) es un patrón de diseño de comportamiento que permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Esto permite que el algoritmo varíe independientemente de los clientes que lo utilizan. En lugar de implementar una lógica condicional compleja dentro de una clase, se aísla cada variación en su propia clase, haciendo el código más flexible y fácil de extender. Se define por tres componentes principales:

- Estrategia: Define una interfaz común a todos los algoritmos soportados. Es el contrato que el cliente utiliza para interactuar con cualquiera de las implementaciones concretas.
- Estrategia Concreta: Son las clases que implementan la interfaz. Cada una contiene una implementación específica de un algoritmo o comportamiento.
- Contexto: Es la clase que utiliza una de las estrategias. Mantiene una referencia a un objeto de la interfaz. Delega la ejecución del algoritmo a este objeto de estrategia. El contexto no sabe qué estrategia específica se está usando; solo sabe que puede llamar al método de la interfaz.

El objetivo es adherirse al Principio Abierto/Cerrado. Elimina la necesidad de usar sentencias de control grandes y difíciles de mantener dentro de la clase de Contexto. Permite cambiar el algoritmo utilizado por el Contexto en tiempo de ejecución. Añadir un nuevo algoritmo o estrategia solo requiere crear una nueva clase que implemente la interfaz Estrategia, sin modificar la clase de Contexto existente. Permite reutilizar los diferentes algoritmos de estrategia con diferentes clases de Contexto.

7. Explicar brevemente el patrón Singleton.

El Patrón Singleton es un patrón de diseño creacional y scope objeto, cuyo objetivo es asegurar que una clase tenga solo una instancia y proporcionar un punto de acceso global a ella. Se caracteriza por tener un constructor privado y un método para acceder a esta única instancia, evitando que los clientes puedan crear múltiples instancias.

8. Definir brevemente el patrón "Composición"

Es un patrón de diseño estructural que permite componer objetos en estructuras de árbol para representar jerarquías construyendo objetos más simples utilizando asociaciones en lugar de herencia. Permite que los clientes traten objetos individuales y composiciones de objetos de manera uniforme. Utiliza la composición recursiva para permitir interactuar con los objetos de manera uniforme sin distinguir entre primitivo y compuesto. Está compuesto por: component (interfaz común), leaf, composite (objeto compuesto) y el cliente.

9. ¿Cuál es la diferencia entre polimorfismo de inclusión y paramétrico?

El polimorfismo de inclusión, también conocido como polimorfismo por subtipado, se basa en la herencia y las interfaces. Permite que un objeto sea tratado como si fuera de cualquiera de sus superclases o interfaces implementadas. En Java, si una variable es declarada de tipo Animal, puede contener una instancia de Perro o Gato. El método llamado es determinado por el tipo real del objeto en tiempo de ejecución (ligadura tardía).

El polimorfismo paramétrico permite definir una estructura, clase o función que puede operar con datos de cualquier tipo sin especificar ese tipo de dato hasta que se le llama o se instancia la estructura. Una única definición de código funciona para múltiples tipos. El código se escribe abstractamente respecto al tipo de dato que maneja. Ejemplo: Las colecciones de Java, como ArrayList<T>.

10. Explicar brevemente el patrón Observer

Es un patrón de comportamiento de objetos que define una dependencia uno-a-muchos de manera que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente. Permite que múltiples vistas se sincronicen con un único modelo de datos sin que estas dependan fuertemente entre sí. Promueve el desacoplamiento entre el sujeto y los observadores. Facilita la escalabilidad, ya que se pueden agregar más observadores sin modificar el sujeto

11. Definir referencia dinámica y estática y la relación entre ellas. Una clase expandida puede ser de tipo dinámico? justifica

Por el polimorfismo, la referencia estática (o tipo declarado) es el tipo de dato que se usa para declarar la variable de referencia en tiempo de compilación, define el contrato o el conjunto máximo de métodos y atributos que pueden ser invocados sobre la variable. Mientras que la referencia dinámica es el tipo de dato concreto de la instancia de objeto que reside en la memoria, define el comportamiento real de los métodos invocados. El tipo dinámico debe ser compatible con el estático, es decir, debe ser instancia de la clase del tipo estático ya sea propia o no. Por esto mismo, una clase expandida (subclase) siempre es instancia de la superclase que extiende, y puede ser el tipo dinámico de una variable declarada del tipo de la superclase.

12. Explicar mapeo directo. Explicar el principio de singleton choice y la regla de ocultamiento de la información

- Mapeo Directo: Es un principio en el diseño orientado a objetos que busca establecer una correspondencia directa entre las entidades del mundo real y las clases en el software. Es decir, cada concepto o entidad del problema que queremos resolver se representa mediante una clase en nuestro programa. Facilita la comprensión del código y la relación entre las diferentes partes. Además, promueve la reutilización de código, ya que las clases creadas pueden ser utilizadas en diferentes contextos.
- Principio de Singleton Choice: establece que una clase debe tener una única responsabilidad bien definida. Al seguir este principio, se logra una mayor cohesión en las clases y se reduce la complejidad del sistema.
- Regla de Ocultamiento de la Información: sugiere que los detalles de implementación de una clase deben ser ocultados a otras clases. Es decir, cada clase debe exponer solo la interfaz necesaria para su uso, y los detalles internos de su funcionamiento deben ser encapsulados

  Ambos principios están estrechamente relacionados y contribuyen a un buen diseño orientado a objetos. Al seguir el principio de singleton choice, se logra una mejor modularidad y se facilita el ocultamiento de la información. Al ocultar la información, se protege la integridad de la clase y se reduce el acoplamiento entre las diferentes partes del sistema.

13. La Confiabilidad del software se define como la suma de dos factores de calidad, explique brevemente cuales son

Corrección: capacidad del software para cumplir con los requisitos funcionales especificados. Los resultados producidos por el software son exactos y confiables. El software se comporta de manera consistente en diferentes situaciones.

Robustez: El software detecta y maneja errores de manera adecuada, evitando bloqueos o comportamientos inesperados. Puede continuar funcionando a pesar de fallas en componentes individuales o en el sistema. Resiste ataques y amenazas de seguridad.

14. ¿Qué se entiende por confiabilidad?¿Por qué se dice que el diseño por contrato favorece la confiabilidad de un producto de software?

La confiabilidad en un software se refiere a su capacidad de funcionar correctamente y de manera consistente a lo largo del tiempo. Un software confiable cumple con los requisitos para los que fue diseñado, es robusto ante errores y es fácil de mantener.

El diseño por contrato es una metodología que mejora significativamente la confiabilidad del software. A través de aserciones (precondiciones, postcondiciones e invariantes), establece un "contrato" formal entre los diferentes componentes del sistema. Esto permite detectar errores tempranamente, aumentar la claridad del código, mejorar la robustez y facilitar el mantenimiento.

15. Explique claramente por qué razón la Regla de Pocas Interfaces favorece el criterio de Continuidad Modular. ¿Favorece esta misma regla algún otro criterio de modularidad?

¿Por qué razón se dice que el Principio de Acceso Uniforme es un caso especial de Ocultamiento de información?

La regla de Pocas Interfaces establece que un módulo (clase o componente) debe exportar o exponer la menor cantidad posible de elementos a sus clientes. Reduce las dependencias entre módulos y aumenta su independencia, lo que hace que el sistema sea más resistente a cambios y más fácil de evolucionar, favoreciendo la continuidad modular.

Sí, esta regla también favorece otros criterios de modularidad como: la cohesión, al limitar las responsabilidades de un módulo, se aumenta su cohesión. El acoplamiento, al reducir el número de interfaces, los módulos están menos interconectados y son más independientes.

16. La diferencia entre patrones de clases y patrones de objetos

La diferencia fundamental entre los Patrones de Clase y los Patrones de Objeto radica en qué manipulan y cuándo se establecen las relaciones (tiempo de compilación vs. tiempo de ejecución). Los Patrones de clases se enfocan en las relaciones estáticas entre clases , definidas en tiempo de compilación. Ejemplos: Factory Method, Abstract Factory. Mientras que los Patrones de objetos se centran en el comportamiento dinámico de los objetos, definidos en tiempo de ejecución. Ejemplos: Singleton, Prototype.

17. ¿Qué mecanismo brinda java para la protección modular?

Java ofrece los modificadores de acceso que controlan la visibilidad de un miembro (clase, método o atributo), niveles de agrupación como paquetes que agrupan clases relacionadas y establecen fronteras de protección modular entre estas, las clases internas que permiten anidar clases dentro de otras aumentando la encapsulación, y las Interfaces y clases abstractas que definen contratos y promueven la abstracción.

18. Smells in code, explicar que es, como solucionarlo, dar 5 ejemplo

Un Code Smell describe una característica o patrón en el código que sugiere la existencia de un problema de diseño. No es un bug ni un error de sintaxis, ya que el código puede funcionar perfectamente. Indica que la estructura del código está dificultando su mantenimiento, legibilidad o extensibilidad, incrementando la deuda técnica. Se soluciona con el Refactoring gradual.

19. Explique claramente la relación entre el principio de sustitución de Liskov y el diseño por contrato.

Principio de Sustitución de Liskov (LSP): Establece que las subclases deben poder sustituir a sus clases base sin alterar el comportamiento esperado del programa. Esto implica que las subclases no deben violar las expectativas de uso establecidas por las clases base.

Diseño por Contrato: Es un enfoque formal que define los acuerdos entre un componente y su entorno, usando precondiciones, postcondiciones e invariantes. Esto asegura que las clases cumplan con ciertos compromisos sobre cómo deben comportarse

Para que una subclase cumpla con el LSP, debe respetar el contrato definido por su clase base. Por ejemplo, una subclase no puede debilitar las precondiciones, ya que esto significa que el comportamiento esperado podría romperse. Asimismo, las postcondiciones de la

subclase deben ser al menos tan estrictas como las de la clase base, garantizando que el contrato sigue siendo válido después de cualquier operación.

20. Explique qué alternativas pueden ofrecer los lenguajes de programación en el tratamiento de las colisiones de nombre.

Una colisión de nombre ocurre cuando dos o más componentes de software (variables, funciones, clases o módulos) tienen el mismo identificador y coexisten en el mismo ámbito o en un ámbito accesible, generando ambigüedad o errores. Los lenguajes de programación ofrecen diversas alternativas para gestionar y prevenir las colisiones de nombre:

- Ámbitos y Visibilidad (Scoping): La herramienta más básica. Los lenguajes definen reglas sobre dónde es visible un identificador (ámbito local, ámbito de clase, ámbito global).
- Paquetes, Módulos o Namespaces: Mecanismos que agrupan lógicamente un conjunto de clases, funciones y otros elementos bajo un nombre contenedor único.
- Sobrecarga de Métodos: Permite que múltiples funciones o métodos dentro de la misma clase o ámbito tengan el mismo nombre, siempre y cuando sus listas de parámetros sean diferentes.
- Renombrado y Alias: Permite que el programador asigne un nombre temporal (alias) a un elemento importado para resolver una colisión.

21. ¿Qué es el Polimorfismo y la Vinculación Dinámica de Código?

El polimorfismo es un pilar de la programación orientada a objetos que permite que objetos de diferentes tipos puedan ser tratados como si fueran del mismo tipo en ciertas circunstancias. Esto se logra gracias a la herencia y a la redefinición de métodos en las subclases. Permite que una única interfaz (un método o una variable de tipo base) pueda ser utilizada para acceder a diferentes implementaciones subyacentes.

La vinculación dinámica es el mecanismo que hace posible el polimorfismo. Es el proceso de determinar la implementación exacta de un método en tiempo de ejecución, en lugar de hacerlo en tiempo de compilación. Cuando se invoca un método en un objeto, el compilador busca el método correspondiente en la clase del objeto. Si el método no se encuentra en esa clase, se busca en las clases padre. El método que finalmente se ejecuta es el más específico para el tipo del objeto en tiempo de ejecución.

22. ¿En que favorece el Paradigma Orientado a Objetos al desarrollo de sistemas de software de alta calidad?

El Paradigma Orientado a Objetos favorece el desarrollo de sistemas de software de alta calidad al proporcionar mecanismos estructurales y de diseño que promueven la Mantenibilidad, la Flexibilidad y la Confiabilidad del código. Esto se logra principalmente mediante la aplicación de sus cuatro pilares esenciales: Encapsulamiento, Abstracción, Herencia y Polimorfismo. Proporciona un diseño modular, donde las partes del sistema son independientes, intercambiables y centradas en responsabilidades específicas. Esto mejora la extensibilidad, y mantenibilidad, permitiendo adaptar el software a cambios sin afectar otras partes inesperadamente. Además, su enfoque refleja el mundo real, facilitando la modelización y promoviendo prácticas que garantizan sistemas robustos, eficientes.

Cada uno de los pilares de POO facilita el diseño modular y el cumplimiento de principios como Abierto/Cerrado, el principio de Sustitución de Liskov, la regla de Ocultamiento de Información, entre muchos otros, imprescindibles para un buen diseño.

23. ¿Qué es el patrón COMMAND?

El Patrón Command es un patrón de diseño de comportamiento que encapsula una solicitud como un objeto, permitiendo parametrizar a los clientes con diferentes solicitudes, encolar o registrarlas, y soportar operaciones que pueden deshacerse. Se identifican 5 componentes: Command (define la interfaz a ejecutar), ConcreteCommand (implementa la interfaz, asociando un objeto receptor a una acción), Client (crea objetos ConcreteCommand y define su receptor), Invoker (llama al metodo execute() para llevar a cabo una solicitud), y Receiver (sabe como realizar las operaciones solicita)

24. ¿Cual es la diferencia entre Concurrencia y Paralelismo?

La Concurrencia y el Paralelismo son conceptos relacionados con la ejecución de múltiples tareas, pero se distinguen por cómo y dónde se ejecutan esas tareas. La concurrencia se basa en tratar múltiples tareas, gestionar múltiples flujos de control, dando la ilusión de ejecución simultánea. Puede lograrse con un solo núcleo de CPU, conmutando rápidamente entre tareas (time-slicing). Busca mejorar la capacidad de respuesta y la gestión de tareas.

Mientras que en el paralelismo las tareas se ejecutan simultáneamente en diferentes núcleos de CPU. Busca mejorar la velocidad de procesamiento (rendimiento) de los cálculos intensivos. Es una propiedad del hardware

25. Explicar en qué consiste el principio de preservación

El Principio de Preservación es un criterio fundamental en el diseño de software, especialmente asociado a la metodología de Diseño por Contrato. Consiste en la obligación de un módulo de no alterar la integridad del estado de un objeto si la rutina falla y no puede completar su tarea. Se refiere a la idea de que los cambios en el código deben minimizar el impacto en el comportamiento existente. Esto es fundamental para mantener la estabilidad y confiabilidad del software a lo largo del tiempo. Los principios SOLID proporcionan herramientas y guías para lograr este objetivo al fomentar diseños modulares, extensibles y mantenibles.

26. ¿Cuáles son las ventajas y desventajas de implementar una relación entre clases por asociación y por herencia?

La asociación describe una relación entre dos clases donde una clase utiliza a otra, pero no depende de ella jerárquicamente. Modela la relación “tiene un”. Ventajas: flexibilidad, permite que la clases trabajen juntas sin una estructura rígida como la jerarquía, facilitando los cambios. Hay Bajo Acoplamiento, porque las clases solo interactúan a través de interfaces bien definidas y permite cambiar la implementación del objeto asociado en tiempo de ejecución (favorece el Principio Abierto/Cerrado - OCP). Permite construir sistemas más modulares y escalables, donde las relaciones pueden cambiar sin modificar el comportamiento de las clases involucradas. Desventajas: dificulta el diseño, puede ser necesario implementar interfaces o clases intermedias, lo que incrementa la complejidad. Si

la asociación no está bien definida, puede provocar que las clases asuman responsabilidades que no deberían.

La herencia establece una relación jerárquica entre clases, donde una subclase hereda propiedades y comportamientos de una clase base. Ventajas: Las clases derivadas pueden reutilizar el código de la clase base, evitando la duplicación. Facilita la extensión del comportamiento de una clase sin modificar la clase base. Facilita la organización. Las clases derivadas pueden ser tratadas como instancias de su clase base, lo que permite el uso de polimorfismo y hace que el código sea más flexible. Desventajas: con un alto acoplamiento se dificulta la modificación de las superclases, causando problemas en las subclases. La estructura es rígida, limitando la flexibilidad.

27. ¿Qué es un trusted component? ¿A qué llama ABCDE de los trusted components y en qué consisten?

Un Componente de Confianza es una unidad de software que es fundamental para la correctitud y la integridad del sistema. Se asume que un componente es de confianza si su diseño y su implementación son tan rigurosos que se puede garantizar que: Siempre operará de acuerdo con su contrato. Si un cliente cumple con su parte del contrato, el componente siempre cumplirá con sus garantías. Un Trusted Component es aquel que ha sido diseñado y verificado para minimizar o eliminar la posibilidad de fallos internos, siendo la base de la confiabilidad del sistema en su conjunto.

El ABCDE es un acrónimo o conjunto de reglas propuesto para describir los cinco criterios que una rutina (o componente) debe cumplir para ser considerada de confianza, en el contexto del Diseño por Contrato y la gestión de excepciones. Estos criterios aseguran que la rutina maneje tanto la corrección de su propia implementación como la respuesta ante errores del cliente o fallos del entorno:

- Assertion (Aserción / Contrato): El componente debe tener un contrato formal bien definido que especifique su funcionamiento y sus garantías.
- Bug-Free: Si el cliente cumple con la precondición, la implementación del código no debe fallar y debe garantizar el cumplimiento de la postcondición. Se asume que el código está implementado correctamente.
- Consistent (Consistente / Preservación): Si el método termina, debe asegurar que se mantenga la Invariante de Clase. Esto se relaciona con el Principio de Preservación: no dejar el objeto en un estado inconsistente tras un fallo.
- Defensive (Defensivo / Detección): La rutina debe ser capaz de detectar si su precondición fue violada por el cliente.
- Exception Handling: La rutina debe tener un manejador de excepciones para actuar ante errores internos o de entorno, siguiendo una política disciplinada.

28. ¿Cumple el lenguaje java con el principio de acceso uniforme?

El Principio de Acceso Uniforme establece que el código cliente debería poder acceder a un valor sin tener que saber si ese valor es un campo almacenado directamente o si es el resultado de un cálculo. La sintaxis de acceso debería ser la misma para ambos casos. La sintaxis de java no permite esto exactamente, pues si se quiere obtener un atributo se le puede pedir a la clase directamente separado de los métodos. Pero, esto se logra siguiendo el principio de Ocultamiento de la información. Haciendo uso de los modificadores de

privacidad que ofrece java, se ocultan los atributos de la clase cliente y se ofrecen métodos getters para que obtengan estos valores, independientemente de si están seteados o calculados.

29. Explique porque el framework struts MVC

El framework Apache Struts es un marco de desarrollo web para Java que fue uno de los primeros en adoptar y formalizar el patrón arquitectónico Modelo-Vista-Controlador (MVC) para aplicaciones web. Se dice que Struts es un framework MVC porque fuerza la separación de las tres responsabilidades del patrón en componentes distintos y bien definidos dentro de la aplicación.

- Modelo: Gestiona la lógica de negocio, el estado de la aplicación y el acceso a los datos
- Vista: Es responsable de la presentación de los datos al usuario y de la captura de la interacción del usuario. No contiene lógica de negocio.
- Controlador: Actúa como intermediario. Recibe las solicitudes del usuario, traduce la entrada a acciones del Modelo y selecciona la Vista apropiada para mostrar el resultado.