Resumen de Cursada y Cuestionario de Promoción - TDP

Este documento reúne la totalidad de los contenidos de la materia Tecnología de Programación (TDP). Está dividido estrictamente en dos partes: primero, el resumen detallado y conceptual de la cursada regular (Clases 04 a 29) enriquecido con los marcos lógicos de ingeniería de software; segundo, el banco de 29 preguntas de promoción.

Parte 1: Resumen de Cursada (Clases 04 a 29)

Clase 04: Principios de Modelado y Modularidad

Modelar es simplificar la realidad para comprender un sistema de software complejo, reduciendo costos y minimizando riesgos de error. En la ingeniería de software, se define al "usuario" tanto como al operador del sistema como al comprador que contrata su desarrollo. Los factores de calidad (correctitud, robustez, extendibilidad, reusabilidad, eficiencia, legibilidad) se garantizan mediante una fuerte modularidad, bajo el principio de que los sistemas deben construirse a partir de componentes cohesivos y débilmente acoplados. Bertrand Meyer establece cinco criterios esenciales de modularidad y cinco reglas de diseño indispensables:

Criterios de Modularidad

- **Descomposición modular:** Dividir un problema complejo en subproblemas autónomos más simples que puedan resolverse de forma independiente.
- **Composición modular:** Combinar libremente elementos de software existentes para crear nuevos sistemas (reutilización).
- **Continuidad modular:** Asegurar que un cambio pequeño en la especificación afecte a uno o a muy pocos módulos.
- **Protección modular:** Contener condiciones anormales (errores) en ejecución dentro de un módulo, evitando que se propaguen catastróficamente.
- **Comprensión modular:** Permitir al desarrollador entender el funcionamiento de un módulo de manera aislada, sin necesidad de conocer el resto del sistema.

Reglas de Diseño Modular

1. **Pocas interfaces:** Reducir la cantidad de vías de comunicación entre módulos para mantener el acoplamiento al mínimo.

2. **Interfaces pequeñas:** Intercambiar la menor cantidad posible de información en cada comunicación.
3. **Interfaces explícitas:** Toda conversación entre módulos debe ser clara y declarada, eliminando dependencias ocultas o efectos laterales.
4. **Ocultamiento de información (Encapsulado):** Cada módulo debe exportar una interfaz clara y mantener sus detalles de implementación estrictamente privados.
5. **Mapeo directo:** Mantener una correspondencia clara y directa entre los conceptos del mundo real y las estructuras de software correspondientes.

Clase 05: Conceptos Fundamentales de la POO

El paradigma de objetos cumple las reglas de Meyer al proponer los siguientes pilares operativos:

- **Clase:** Descripción estática que actúa simultáneamente como módulo y tipo de datos, especificando atributos (estado) y operaciones (servicios).
- **Objeto:** Instancia dinámica de una clase que interactúa mediante paso de mensajes. Existe una única *instancia actual* en ejecución en cada instante (`this` o `self`).
- **Relaciones lógicas:** Asociación (vínculo de conocimiento), Herencia (relación "es un"), e Instancias propias (objetos de la misma clase) vs de descendientes (objetos de subclases).
- **Polimorfismo y Ligadura Dinámica:** Capacidad de enviar mensajes idénticos a objetos de tipos distintos. La ligadura dinámica determina en tiempo de ejecución el método exacto basándose en el tipo dinámico del objeto real y no en el tipo estático de la referencia.

Clase 06: Survey de Diferentes Lenguajes Orientados a Objetos

Las ideas de la POO gobiernan todo el proceso de construcción del software; el lenguaje es una herramienta de implementación que ofrece mecanismos técnicos para plasmar el diseño bajo filosofías distintas:

- **Lenguajes Basados en Clases (Java, C++, Smalltalk):** Exigen la definición estática de una clase para actuar como molde estructural invariable del cual nacerán las instancias en memoria.
- **Lenguajes Basados en Prototipos (JavaScript):** Prescinden de clases rígidas. Los objetos se crean de forma dinámica e independiente y heredan rasgos delegando sus mensajes hacia otros objetos denominados prototipos. Las clases modernas en estos entornos son azúcar sintáctico sobre su cadena nativa de prototipos físicos.

```
// Ejemplo de funciones constructoras en semántica por prototipos
function crearPunto(x, y) {
  this.coordx = x;
```

```
    this.coordy = y;
    this.mostrar = function() { document.write("<" + this.coordx + "," + this.coordy + ">"); };
}
p1 = new crearPunto(2, 3);
p1.mostrar(); // Devuelve <2,3>
```

Clase 07: UML - Diagrama de Clases (Parte 1)

UML es el estándar gráfico para modelar aspectos estáticos y dinámicos. Una clase se compone de tres compartimientos: Nombre, Atributos (`:=`) y Operaciones (`():`), usando visibilidades Pública (`+`), Privada (`-`) y Protegida (`#`). Las relaciones estructurales de agrupación todo-parte se dividen en:

- **Agregación:** Asociación jerárquica con dependencia estructural donde las partes tienen sentido juntas pero pueden sobrevivir de forma independiente al contenedor. Se grafica con un rombo vacío del lado del contenedor.
- **Composición:** Relación restrictiva donde el contenedor controla el ciclo de vida de las partes; las partes no pueden existir separadas del todo. Se grafica con un rombo relleno.

Clase 08: Tratamiento de Objetos, Semánticas y Diagramas de Interacción

La gestión física de memoria plantea dos aproximaciones de semántica de asignación:

Característica	Semántica por Referencia (Java)	Semántica por Valor (C++)
Acceso	A través de punteros lógicos o direcciones de memoria.	Almacenamiento físico directo del subobjeto dentro del contenedor.
Creación	Explícita mediante operadores dedicados (`new`).	Implícita y automática junto con el objeto padre.
Aliasing	Permitido (múltiples referencias al mismo objeto).	Imposible (cada variable contiene su propia copia física).

Se definen operaciones como comparación de referencias (`==`), comparación de objetos (`equals`), copia superficial (`copy`) y clonación (`clone`). Para el comportamiento dinámico, se

dispone del Diagrama de Secuencia (prioriza el orden cronológico vertical) y el Diagrama de Colaboración o Comunicación (prioriza conexiones estructurales usando numeración decimal indexada como `1.1`, `1.1.1`).

Clases 09 a 12: Proceso de Modelado - Caso de Estudio "Base Starcraft"

Taller práctico iterativo para transformar un enunciado en un diseño robusto:

- **Identificación y Asignación de Roles:** Se leen los sustantivos evaluando relevancia por frecuencia. Conceptos circunstanciales se transforman en atributos, mientras que entidades complejas adquieren rol de Clase Autónoma asignándoseles responsabilidades claras.
- **Refinamiento Estático y Validación Dinámica:** Se traslada el modelo a UML. Para la operación `depositarTitanio()`, se modela secuencialmente que la Fábrica exponga sus contenedores lógicos; el Robot evalúa cíclicamente (`loop`) si el componente está `disponiblePara(compartimiento)` y de forma condicional (`alt`) delega el comportamiento ejecutando `recibirCarga(compartimiento)`, logrando un diseño desacoplado.

Clase 13: Generalización y Herencia Avanzada

La generalización unifica características comunes en superclases. La visibilidad protegida (`#`) restringe el acceso a miembros para el entorno exterior, pero permite que las clases hijas accedan de forma directa. La redefinición (overriding) permite a la subclase declarar un método heredado modificando su algoritmo pero manteniendo intacta su firma, utilizando la palabra clave `super` para invocar la versión de la superclase y reutilizar o complementar su lógica.

Clase 14: Polimorfismo, Chequeo y Conformidad de Tipos

El sistema opera diferenciando el Tipo Estático (fijo, usado para declarar la entidad en el código fuente y controlado en compilación) del Tipo Dinámico (variable, determinado por la clase del objeto real asignado en memoria). Una asignación polimórfica es legítima si el tipo de la derecha presenta Conformidad de Tipos respecto al tipo estático de la izquierda, actuando como descendiente formal. Combinando colecciones genéricas de tipo estático base común con instancias dinámicas heterogéneas conformes, se configuran las estructuras de contenido polimórfico.

Clase 15: Tipos, Subtipos y Herencia de Interfaces

Un tipo de datos se refiere al "qué hace" (interfaz o especificación pública), mientras que la clase define el "cómo lo hace" (implementación o estado interno). La herencia de interfaz asegura conformidad de tipos bajo el Principio de Sustitución, mientras que la herencia de

clase comparte código pero puede sufrir del problema de la superclase frágil si cambios internos en el padre rompen a las hijas. Las Clases Abstractas son abstracciones incompletas con métodos abstractos/diferidos, no toleran la instanciación directa y actúan como contratos obligatorios para sus subclases concretas.

Clases 16 a 18: Catálogo de Patrones de Diseño (GoF)

Los patrones GoF ofrecen soluciones probadas a problemas recurrentes comunicantes, divididos según su propósito:

- **Creacionales (Abstraen la instanciación):** *Abstract Factory* (interfaz para crear familias de objetos relacionados sin especificar clases concretas), *Prototype* (genera instancias clonando un espécimen modelo en memoria), y *Singleton* (asegura una única instancia con acceso global).
- **Estructurales (Organización de clases y objetos):** *Adapter* (Wrapper que convierte una interfaz incompatible en otra esperada por el cliente), *Composite* (compone objetos en árboles para tratar igual a elementos individuales y compuestos), y *Decorator* (anexa responsabilidades a un objeto de forma dinámica en ejecución).
- **De Comportamiento (Algoritmia y asignación de responsabilidades):** *Command* (encapsula una petición como un objeto independiente para encolar o deshacer), *Observer* (dependencia uno-a-muchos para notificar cambios automáticamente desacoplando sujeto y observadores), *Strategy* (familia de algoritmos intercambiables en ejecución de scope Object), *State* (delega el comportamiento en objetos estados según el estado interno), y *Visitor* (define operaciones sobre una estructura sin modificar sus clases mediante doble ligadura).

Clases 19 y 20: Taller de Modelado - Caso "Road Fighter" (Parte 1)

Análisis de un videojuego para ejercitar taxonomías avanzadas e interacciones complejas. Los móviles se desglosan bajo la superclase *Vehiculo*, divididos en *Carrera* (con lógica de competencia, conteniendo al Jugador) y *Transito* (obstáculos dinámicos). Se incorporan los Obstaculos fijos (Anciana, Perro, Bache y la extensión del Lobo que duplica el peso y reinicia el puntaje) y PowerUps. Para eludir el acoplamiento cruzado de impactos, se diseñan las interfaces funcionales `Colisionador` y `Colisionable`, delegando la lógica de colisión en el componente impactado en la ruta.

Clase 21: Patrón Arquitectónico de Tres Capas

Patrón de diseño jerárquico unificado que propone estructurar la aplicación organizando el código en tres niveles independientes con una fuerte separación de responsabilidades:

Capa	Responsabilidades Principales
Vista (Presentación)	Se encarga de la visualización de la información e interacción con el usuario, traduciendo acciones en eventos enviados a la lógica.
Lógica (Negocio)	Se encarga de procesar la información, aplicar las reglas de negocio e interpretar las solicitudes. Actúa como intermediaria pura.
Datos (Persistencia)	Se encarga del almacenamiento, recuperación y actualización, abstrayendo los detalles específicos de las fuentes de datos.

El motor directo de esta división es lograr un bajo acoplamiento, aislando bugs y permitiendo que cada nivel escale o se modifique de forma independiente sin afectar la lógica central.

Clase 22: Aplicación de Arquitectura - Caso "Road Fighter" (Parte 2)

Se segmenta el proyecto en los paquetes independientes `vista` (Java Swing, paneles de dibujo) y `logica` (clase central `Juego` encargada de coordenadas matemáticas y reglas lógicas). La comunicación se media mediante interfaces de control (`ControladorJuego`). Para plasmar los movimientos de la lógica en la pantalla de forma desacoplada, se implementa el patrón **Observer**, donde las vistas actúan como observadores de las entidades lógicas y actualizan los gráficos ante notificaciones de cambio.

Clase 23: Fundamentos de Concurrencia y Multiprocesamiento

La concurrencia gestiona múltiples flujos de control compartiendo el espacio de memoria (intercalando tareas mediante time-slicing en un núcleo), mientras que el paralelismo es la ejecución física simultánea en múltiples procesadores de hardware. Java permite multithreading heredando de `Thread` o implementando `Runnable` mediante los métodos `run()` y `start()`. Para asegurar exclusión mutua y código seguro para hilos (thread-safety), se utiliza la palabra clave `synchronized`, obligando al hilo a tomar el *lock* del objeto antes de ingresar al bloque.
Los riesgos críticos son las condiciones de carrera o Data Race (modificación simultánea

corrupta) y el **Interbloqueo (Deadlock)**, que ocurre cuando hilos se bloquean indefinidamente esperando recursos que el otro retiene de forma cruzada bajo las condiciones de Coffman. Se previene aplicando un **Orden Estricto de Locks**: obligar a todos los hilos a adquirir los locks en un orden predefinido global unificado, rompiendo el ciclo de espera.

Clase 24: Concurrencia Práctica - Caso "Road Fighter" (Parte 3)

El bucle principal de actualización del juego (avanzar la ruta, mover los autos de tránsito automáticos y recalcular colisiones matemáticas de forma constante) se confía a un **Thread concurrente dedicado e independiente** que corre de manera asíncrona a la captura del teclado. Este hilo ejecuta actualizaciones periódicas reguladas mediante `Thread.sleep(delay)`, garantizando un movimiento fluido en tiempo real e independiente de la interacción del usuario.

Clase 25: Mecanismos de Serialización y Configuración

La persistencia se complementa mediante la **Serialización**, proceso que transforma un objeto vivo en memoria en un flujo de bytes binarios para guardarlo en archivos físicos de disco o transmitirlo por red, logrando recuperarlo mediante deserialización. La parametrización se delega en archivos de configuración externos independientes tipo **Properties** (estructuras clave-valor), permitiendo alterar constantes lógicas sin necesidad de recompilar la aplicación.

Temas Complementarios: Refactoring, Contratos y Excepciones

Refactoring y Code Smells: Consiste en cambiar la estructura interna del código de forma disciplinaria para aumentar su legibilidad y mantenibilidad sin alterar el comportamiento externo. Los Code Smells son indicios o alarmas de problemas de diseño. Se aplica en un ciclo continuo: 1. Identificar el Smell, 2. Crear pruebas unitarias de seguridad, 3. Realizar cambios graduales pequeños, 4. Verificar con las pruebas y 5. Repetir.
Diseño por Contrato (DbC): Entiende las relaciones entre clases y clientes como un acuerdo formal expresado mediante aserciones (fórmulas de correctitud de la forma `{P} A {Q}`). Consta de Precondiciones (propiedades obligatorias para llamar a la operación), Postcondiciones (garantías que asegura el método al retornar) e Invariantes de clase (restricciones generales válidas en todo estado estable). En herencia, una redefinición de método debe respetar el contrato del padre: incluir una precondición igual o más débil y una postcondición igual o más estricta (LSP). Java provee la sentencia `assert` para detectar bugs locales en desarrollo, pero es desactivable en entornos de producción y no establece contratos formales de responsabilidades.
Gestión Disciplinada de Excepciones: Se diferencia el Error (decisión equivocada en desarrollo), el Defecto (característica del software que puede fallar) y la Falla (desviación de objetivos en ejecución). Ante un fallo irrecuperable de entorno, se aplica la estrategia del **Pánico Organizado**: fallar de manera controlada y propagar o relanzar la excepción hacia el

llamador para notificar la ruptura del contrato de forma predecible sin dejar el objeto inconsistente (Principio de Preservación). Java provee bloques `try-catch`, `finally` (ejecución incondicional de limpieza) y `throws`. Las excepciones chequeadas (checked) representan condiciones externas recuperables que el compilador obliga a manejar o declarar, mientras que las no chequeadas (unchecked) reflejan errores puros de programación y su manejo es opcional.

Parte 2: Cuestionario de Promoción (29 Preguntas)

Pregunta 1: Analice y explique cuáles son los principales inconvenientes y desventajas al utilizar instanceof.

El principal problema de usar instanceof es que rompe el principio de Abierto/Cerrado (OCP) de SOLID y va en contra del polimorfismo. Cuando usás instanceof, estás revisando el tipo exacto de un objeto en medio de la ejecución para decidir qué hacer, lo que te obliga a llenar el código de condicionales (if/else). Si mañana agregás una clase nueva a tu jerarquía, vas a tener que ir a buscar y modificar todos esos condicionales a mano. Esto genera mucho acoplamiento y hace que el mantenimiento sea difícil. Lo ideal es usar polimorfismo puro (definir el método en la interfaz común y que cada subclase haga lo suyo) o patrones como Visitor y colecciones genéricas.

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

Esto pasa gracias a dos pilares de la POO: la herencia y el polimorfismo. Por un lado, la herencia crea una relación del tipo "es un". Si la clase Perro hereda de Animal, un objeto de tipo Perro es un Perro, pero por herencia también se lo considera un Animal. Por otro lado, el polimorfismo permite que una variable de tipo general (superclase o interfaz) guarde una referencia a un objeto más específico en la memoria. Así, aunque en la memoria tengas un objeto real Gato, tu variable de referencia lo puede tratar simplemente como un Animal.

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

Hay cuatro aspectos clave diseñados para maximizarla:

- **La Herencia:** Te permite reutilizar directamente el código (atributos y métodos) que ya escribiste en la clase padre dentro de las clases hijas, sin tener que volver a escribirlo desde cero.
- **La Abstracción (Interfaces y Clases Abstractas):** Definen un contrato común (qué se puede hacer) sin meterse en los detalles de cómo se hace. Esto ayuda a crear componentes genéricos reusables de diseño.
- **El Encapsulamiento:** Al esconder los detalles de cómo funciona una clase por dentro

(ocultación de información) y dejar libre solo su interfaz pública, lográs un bajo acoplamiento. Si cambiás el código interno, los demás módulos no se rompen y pueden seguir usándola.
- **El Polimorfismo:** Te permite tratar a diferentes objetos de forma uniforme y facilita la Inyección de Dependencias, lo que significa que podés reutilizar el mismo código cliente pasándole distintas implementaciones en tiempo de ejecución.

Pregunta 4: Explique cuál es el objetivo del uso de métricas de software. Relacionar con medir factores de calidad. Describa tres métricas importantes y una poco útil.

El objetivo de las métricas es dar datos numéricos y objetivos para medir, monitorear y controlar la calidad del código. Por ejemplo, sirven para medir el factor de mantenibilidad analizando el acoplamiento y la cohesión. Tres métricas importantes en POO son:
- **Número de métodos por clase:** Si una clase tiene demasiados métodos, probablemente tiene demasiadas responsabilidades y le falta cohesión.
- **Acoplamiento entre clases:** Mide qué tanto depende una clase de otras. Si está muy acoplada, es difícil cambiarla o reutilizarla.
- **Profundidad de herencia:** Qué tan lejos está una clase de la raíz de la jerarquía. Si es muy larga, el diseño se vuelve complejo y difícil de seguir.

Una métrica poco útil es el **Número de líneas de código (LOC)**, porque no considera factores como la complejidad algorítmica, la legibilidad o la eficiencia del diseño.

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

El refactoring afecta principalmente a la **Mantenibilidad** y a la **Fiabilidad / Confiabilidad**. Por un lado, mejora muchísimo la Mantenibilidad porque ordena la estructura del código, haciendo que sea más fácil de entender y de modificar en el futuro sin meter errores. Por otro lado, representa un riesgo a corto plazo para la Fiabilidad, porque estás tocando y modificando la lógica de un código que ya funciona, lo que abre la posibilidad de cometer un error humano (regresión). Para mitigar este costo directo, es obligatorio acompañar el proceso con pruebas unitarias robustas que funcionen como red de seguridad.

Pregunta 6: Explique claramente el patrón de diseño Strategy.

El patrón Strategy es un patrón de comportamiento de objetos que sirve para guardar diferentes formas de hacer una misma tarea (algoritmos) en clases separadas y poder intercambiarlas en tiempo de ejecución de manera flexible. En lugar de meter un condicional `if/else` gigante adentro de una clase, aislás cada opción en su propia estructura. Tiene tres componentes:

- **Estrategia:** Una interfaz común que define el contrato que el cliente va a usar.
- **Estrategia Concreta:** Las clases reales que implementan esa interfaz (cada una con su algoritmo específico).
- **Contexto:** La clase que usa la estrategia. Mantiene una referencia a la interfaz y le delega el trabajo, sin importarle cuál estrategia específica se está usando en tiempo de ejecución, cumpliendo con el principio de Abierto/Cerrado (OCP).

Pregunta 7: Explicar brevemente el patrón Singleton.

El patrón Singleton es un patrón creacional de scope objeto cuyo objetivo es asegurarse de que una clase tenga una única instancia en todo el programa y proporcionar un punto de acceso global a ella. Para lograrlo, la clase tiene su constructor privado (así los clientes no pueden crear múltiples instancias con `new`) y ofrece un método estático público que se encarga de crear la instancia la primera vez o devolver la que ya existe en memoria.

Pregunta 8: Definir brevemente el patrón "Composición" (Composite).

Es un patrón de diseño estructural que permite componer objetos en estructuras de árbol para representar jerarquías todo-parte construyendo objetos más complejos utilizando asociaciones en lugar de herencia. Su ventaja es que permite que los clientes traten a objetos individuales simples (hojas/leaf) y a composiciones de objetos combinados (compuestos) de manera uniforme y homogénea, usando la composición recursiva para interactuar con ellos sin tener que andar distinguiendo entre primitivo y compuesto.

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

La diferencia radica en el mecanismo técnico utilizado:

- **Polimorfismo de inclusión (por subtipado):** Se basa en la herencia y las interfaces. Permite que un objeto sea tratado como si fuera de cualquiera de sus superclases. El método exacto que se ejecuta se determina en tiempo de ejecución según el objeto real en memoria (ligadura tardía).
- **Polimorfismo paramétrico (Genericidad):** Permite definir una estructura, clase o función que puede operar con datos de cualquier tipo sin especificar ese tipo real hasta el momento en que se la llama o se la instancia. El código se escribe abstractamente respecto al tipo de dato que maneja (ejemplo: las colecciones como `ArrayList<T>`).

Pregunta 10: Explicar brevemente el patrón Observer.

Es un patrón de comportamiento de objetos que define una dependencia uno-a-muchos. Cuando el objeto principal (Sujeto o Modelo) cambia su estado, todos sus dependientes registrados (Observadores o Vistas) son notificados y actualizados automáticamente. Esto promueve el desacoplamiento y permite que múltiples vistas se sincronicen con un único

modelo sin depender fuertemente entre sí, facilitando agregar más observadores sin tocar el sujeto.

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

Por el polimorfismo, la **referencia estática (tipo estático)** es el tipo de dato que se usa para declarar la variable en tiempo de compilación; define el contrato o conjunto máximo de métodos que pueden ser invocados. La **referencia dinámica (tipo dinámico)** es el tipo de dato concreto del objeto real que reside físicamente en la memoria; define el comportamiento real de los métodos ejecutados. Relación: el tipo dinámico debe ser compatible con el estático (debe ser de la misma clase o una subclase). Por esto mismo, una clase expandida (subclase) siempre es una instancia de la superclase que extiende, por lo que **sí puede ser el tipo dinámico** de una variable declarada con el tipo de la superclase.

Pregunta 12: Explique mapeo directo. Explique el principio de singleton choice y la regla de ocultamiento de la información.

Mapeo Directo: Es un principio de diseño que busca establecer una correspondencia directa entre las entidades del mundo real y las clases en el software, facilitando la comprensión y reutilización del código. El **Principio de Singleton Choice** establece que una clase debe tener una única responsabilidad bien definida, logrando una mayor cohesión y reduciendo la complejidad. La **Regla de Ocultamiento de la Información** sugiere que los detalles de implementación interna de una clase deben ser privados, exponiendo solo la interfaz pública necesaria para su uso. Ambos principios están relacionados: al seguir el singleton choice, se logra una mejor modularidad, lo que facilita proteger la integridad de la clase y reducir el acoplamiento con otras partes del sistema.

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

Se compone de la suma de:

- **Corrección:** La capacidad del software para cumplir con exactitud con los requisitos funcionales especificados, produciendo resultados consistentes en diferentes situaciones.
- **Robustez:** La capacidad del programa para detectar y manejar errores o situaciones anormales de manera adecuada, evitando bloqueos, caídas o comportamientos inesperados.

Pregunta 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 sin fallar. El diseño por contrato es una metodología que mejora significativamente esta confiabilidad porque, a través de aserciones formales (precondiciones, postcondiciones e invariantes), establece un "contrato" explícito entre los diferentes componentes. Esto permite detectar errores de forma muy temprana durante el desarrollo, aumentar la claridad, mejorar la robustez y saber exactamente de quién es la culpa (cliente o proveedor) si ocurre un fallo, facilitando el mantenimiento.

Pregunta 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 una clase debe exponer la menor cantidad posible de elementos a sus clientes. Esto reduce las dependencias y aumenta la independencia de los módulos, lo que hace que el sistema sea más resistente a cambios; si modificás una especificación, el impacto se frena afectando a uno o pocos módulos, favoreciendo directamente la **Continuidad Modular**. Sí, esta regla también favorece la alta cohesión y el acoplamiento débil. Por su parte, el **Principio de Acceso Uniforme** dice que el código cliente debería poder acceder a un valor con la misma sintaxis, sin tener que saber si ese valor está guardado en una variable o si es el resultado de un cálculo. Es un caso especial de ocultamiento de información porque, al hacer uso de los modificadores de privacidad y ofrecer métodos *getters* (`objeto.getValor()`), se ocultan los atributos internos y el cliente accede siempre de forma idéntica y uniforme.

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

La diferencia fundamental radica en qué manipulan y cuándo se establecen sus uniones estructurales:

- **Patrones de clases:** Se enfocan en las relaciones estáticas entre clases y se definen en tiempo de compilación utilizando principalmente la herencia (ejemplos: Factory Method, Abstract Factory).
- **Patrones de objetos:** Se centran en el comportamiento dinámico y la comunicación de las instancias, definiéndose en tiempo de ejecución mediante la técnica de la composición (ejemplos: Singleton, Strategy, Prototype).

Pregunta 17: ¿Qué mecanismo brinda Java para la protección modular?

Java ofrece varias herramientas nativas para cuidar los módulos: los modificadores de acceso (`public`, `private`, `protected` y por defecto) que controlan la visibilidad de los miembros; niveles de agrupación como paquetes que reúnen clases relacionadas y ponen fronteras de protección; clases internas que permiten anidar una estructura dentro de otra aumentando la

encapsulación; e interfaces o clases abstractas que definen contratos puros para promover la abstracción separando el "qué" del "cómo".

Pregunta 18: Smells in code, explicar qué es, cómo solucionarlo y dar 5 ejemplos.

Un Code Smell describe una característica o patrón en el código fuente que sugiere que existe un problema profundo con el diseño del software. No es un bug de ejecución ni un error de sintaxis (el programa puede andar bien), pero te indica que la estructura está enredada, dificultando su mantenimiento, legibilidad o extensión e incrementando la deuda técnica. Se soluciona aplicando Refactoring gradual (cambios chicos e incrementales). Cinco ejemplos comunes son:

1. *Método demasiado largo:* Una función que intenta asumir demasiadas tareas juntas.
2. *Clase con baja cohesión:* Una clase gigante que maneja conceptos que no están relacionados.
3. *Duplicación de código:* Copiar y pegar bloques de código idénticos en varios lados.
4. *Mal nombre de variable:* Identificadores crípticos que no revelan su intención real.
5. *Envidia de características (Feature Envy):* Cuando un método de una clase interactúa y usa más los atributos de una clase externa que los suyos propios.

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

El Principio de Sustitución de Liskov (LSP) exige que las subclases puedan sustituir a sus clases base de forma transparente sin romper el programa ni alterar el comportamiento esperado. Para lograr esto, la subclase debe respetar obligatoriamente el contrato definido por la superclase. Esto significa que la subclase **no puede debilitar las precondiciones** (no puede exigir más requisitos de entrada que el padre, ya que rompería las llamadas de los clientes clientes), y al mismo tiempo, **sus postcondiciones deben ser al menos tan estrictas o fuertes como las de la clase base**, garantizando que el acuerdo siga siendo válido después de la operación y el subtipo sea un sustituto seguro.

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

Una colisión de nombres ocurre cuando dos o más componentes de software comparten el mismo identificador en un ámbito accesible, generando ambigüedad. Los lenguajes ofrecen alternativas para prevenirlo y gestionarlo: los Ámbitos y visibilidad (Scoping locales o de clase); Paquetes, Módulos o Namespaces que agrupan elementos bajo un nombre contenedor único; Sobrecarga de métodos que permite usar el mismo nombre si varían los parámetros; y cláusulas de Renombrado o Alias que permiten asignar un nombre temporal a un elemento

para resolver el choque en la importación o herencia.

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

El **polimorfismo** es un pilar de la POO que permite que objetos de diferentes clases concretas puedan ser tratados como si fueran del mismo tipo general a través de una interfaz o superclase común compartida. La **vinculación dinámica (o ligadura tardía)** es el mecanismo técnico que hace posible este polimorfismo: es el proceso de buscar y determinar la implementación exacta del método recién en **tiempo de ejecución** (en lugar de hacerlo al compilar), revisando la clase real del objeto alojado en memoria para ejecutar el algoritmo más específico.

Pregunta 22: ¿En qué favorece el Paradigma Orientado a Objetos al desarrollo de sistemas de software de alta calidad?

Favorece la alta calidad al proporcionar herramientas estructurales que promueven directamente la Mantenibilidad, Flexibilidad y Confiabilidad del código. Esto se logra mediante la aplicación de sus cuatro pilares esenciales (Encapsulamiento, Abstracción, Herencia y Polimorfismo), que fuerzan un diseño modular donde las partes del sistema son independientes, centradas en una sola responsabilidad, intercambiables y con un acoplamiento débil. Además, al reflejar fielmente los conceptos del mundo real, facilita el modelado y permite adaptar el software a cambios sin romper otras partes inesperadamente.

Pregunta 23: ¿Qué es el patrón COMMAND? Componentes.

El Patrón Command es un patrón de comportamiento que encapsula una solicitud o acción como un objeto autónomo. Esto permite parametrizar clientes con diferentes peticiones, meter acciones en colas o registros, y soportar operaciones que pueden deshacerse (undo/redo). Se compone de 5 participantes:

- **Command:** Define la interfaz común con el método `execute()`.
- **ConcreteCommand:** Implementa la interfaz asociando un objeto receptor específico a una acción interna.
- **Client:** Crea los objetos de los comandos concretos y define cuál va a ser su receptor.
- **Invoker:** Llama al método `execute()` para llevar a cabo la solicitud cuando corresponde.
- **Receiver (Receptor):** El objeto real de negocio que sabe cómo realizar físicamente las operaciones solicitadas.

Pregunta 24: ¿Cuál es la diferencia entre Concurrencia y Paralelismo?

La diferencia radica en el hardware involucrado y cómo se ejecutan las tareas:

- **La Concurrencia:** Se basa en gestionar y tratar múltiples flujos de control al mismo tiempo, dando la ilusión de ejecución simultánea. Se logra compartiendo un único núcleo

de CPU, conmutando rápidamente entre tareas mediante porciones de tiempo (time-slicing) para mejorar la capacidad de respuesta.

- **El Paralelismo:** Las tareas se ejecutan estrictamente de forma física y simultánea al mismo milisegundo exacto. Requiere obligatoriamente diferentes núcleos de CPU independientes corriendo en paralelo; es una propiedad directa del hardware para ganar velocidad en cálculos pesados.

Pregunta 25: Explicar en qué consiste el principio de preservación.

Es un criterio fundamental asociado a la metodología de Diseño por Contrato. Consiste en la obligación de un método de **no alterar ni corromper la integridad del estado de un objeto** si la rutina sufre una falla o excepción y no puede completar su tarea. El objeto debe quedar intacto y limpio en el estado válido en que estaba antes de la llamada, impidiendo estados inconsistentes a medio guardar que degraden la confiabilidad del software.

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

La **Asociación** modela la relación "tiene un" (una clase usa a otra sin depender de ella jerárquicamente). *Ventajas:* Alta flexibilidad, bajo acoplamiento, permite interactuar mediante interfaces mínimas y cambiar el objeto asociado en tiempo de ejecución (favorece el principio Abierto/Cerrado). *Desventajas:* Puede complejizar el diseño al requerir más interfaces intermedias de conexión. Por su parte, la **Herencia** modela la jerarquía rígida "es un". *Ventajas:* Alta reutilización de código directo evitando duplicaciones, facilita la extensión del comportamiento y permite el uso nativo del polimorfismo. *Desventajas:* Alto acoplamiento (cualquier cambio en la superclase puede romper las subclases) y rigidez estructural.

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

Un **Componente de Confianza (Trusted Component)** es una unidad de software tan bien verificada y diseñada que se puede garantizar que siempre operará de acuerdo con su contrato; si el cliente cumple la precondición, el componente jamás fallará internamente. El **ABCDE** describe los 5 criterios que debe cumplir una rutina para ganarse esa confianza en el contexto de Diseño por Contrato:

- **Assertion (Aserción):** El componente debe poseer un contrato formal explícito (precondiciones, postcondiciones e invariantes).
- **Bug-Free:** El código interno está implementado correctamente y no debe fallar, asegurando la postcondición si se cumplió la entrada.
- **Consistent (Consistente):** Si el método termina (sea por éxito o por fallo), debe asegurar que se mantenga la Invariante de Clase (Principio de Preservación).

- **Defensive (Defensivo):** La rutina debe ser capaz de detectar activamente si el cliente violó la precondición antes de operar.
- **Exception Handling:** Debe tener un manejador de excepciones disciplinado para actuar de forma controlada ante fallos imprevistos del entorno.

Pregunta 28: ¿Cumple el lenguaje Java con el principio de acceso uniforme?

A nivel de sintaxis estricta, **no**, porque Java diferencia gramaticalmente el acceso a una variable de instancia (`objeto.valor`) de la llamada a un método (`objeto.valor()`). Sin embargo, **se logra cumplir a nivel de diseño mediante el Ocultamiento de la información**: al usar los modificadores de acceso para hacer todos los atributos privados y ofrecer exclusivamente métodos *getters* públicos (`objeto.getValor()`), el cliente accede siempre de forma unificada utilizando la sintaxis de métodos, sin tener idea de si el valor está guardado directamente en un campo o si es resultado de un cálculo interno.

Pregunta 29: Explique por qué el framework Struts adopta el patrón arquitectónico MVC.

El framework Apache Struts adopta el patrón Modelo-Vista-Controlador (MVC) para aplicaciones web porque fuerza e impone una separación estricta de las tres grandes responsabilidades del sistema en componentes independientes: el **Modelo** (gestiona la lógica del negocio, las reglas de validación y el acceso a los datos de backend); la **Vista** (responsable puramente de la presentación gráfica y captura de clicks de usuario en el frontend, sin contener lógica de negocio); y el **Controlador** (intermediario que recibe las solicitudes HTTP, traduce las entradas en acciones del Modelo y selecciona dinámicamente cuál Vista apropiada mostrarle en pantalla al usuario con el resultado final).