**S08.s2 - Principio de diseño Program to an interface  
Conocimientos previos**

**¿Qué diferencia hay entre una interfaz y una clase abstracta en Java?**

**Interfaz:**

**Define un contrato de métodos sin implementación.**

**Una clase puede implementar múltiples interfaces.**

**No tiene estado ni constructores.**

**Usada para desacoplar y definir comportamientos comunes sin compartir implementación.**

**Clase Abstracta:**

**Proporciona una implementación parcial y puede tener estado (campos) y constructores.**

**Una clase solo puede extender una clase abstracta.**

**Usada para compartir lógica común y estado entre clases relacionadas.**

**¿Has trabajado previamente con el principio de diseño "Program to an**

**Interface, not an Implementation" en tus proyectos de desarrollo de**

**software? Si es así, ¿puedes proporcionar un ejemplo o una experiencia**

**relacionada con este principio?**

**Objetivo: Programar contra el contrato (interfaz), no contra una implementación concreta.**

**Beneficios:**

**Desacoplamiento: el código depende de interfaces, no de implementaciones específicas.**

**Flexibilidad: puedes cambiar la implementación sin afectar el resto del sistema.**

**Testabilidad: permite usar implementaciones simuladas para pruebas (como mocks o stubs).**

**UTILIDAD**

**¿Cómo podría este principio de diseño mejorar la flexibilidad del código en**

**un proyecto de desarrollo de software en Java?**

**El principio de programar contra una interfaz en lugar de una implementación concreta mejora la flexibilidad de varias maneras:**

**Desacoplamiento: Al depender de interfaces en lugar de clases concretas, el código se desacopla de las implementaciones.**

**Sustitución fácil de implementaciones: Puedes reemplazar una implementación por otra sin tener que cambiar el código que la usa.**

**Mayor escalabilidad: Si en el futuro decides agregar nuevas funcionalidades (como una notificación por push o integración con un servicio de mensajería instantánea)**

**¿En qué medida podría facilitar la incorporación de nuevas características o**

**cambios en el software existente?**

**Pruebas:**

**Facilita el uso de mocks y stubs: Cuando el código depende de interfaces, es mucho más fácil usar mocks y stubs en pruebas unitarias.**

**Mantenimiento:**

**Reducción de la deuda técnica: La separación de la lógica de negocio y la implementación reduce la probabilidad de acoplamiento fuerte entre componentes, lo cual disminuye la deuda técnica**

**¿Qué ventajas podría ofrecer en términos de pruebas y mantenimiento de**

**código en comparación con no seguir este principio?**   
**Mayor acoplamiento: El código se volverá dependiente de implementaciones concretas, lo que significa que los cambios en una implementación podrían afectar muchas otras partes del sistema, haciendo el mantenimiento más costoso y arriesgado.**

**INTRODUCCIÓN**

**El principio de diseño "Program to an interface, not an implementation" es un concepto clave en el mundo de la programaciónorientada a objetos y se utiliza comúnmente en Java. Este principio se refiere a la idea de que, en lugar de depender deimplementaciones concretas de clases, deberías programar en función de interfaces o abstracciones más generales. Estopromueve la flexibilidad y la capacidad de extensión de tu código.**

**USAR INTERFACES COMO TIPOS**

**En lugar de declarar variables o parámetros con tipos de clases concretas, puedes utilizar interfaces como tipos. Esto permite que tu código sea más flexible y se pueda adaptar a diferentes implementaciones en tiempo de ejecución.**

**INYECCIÓN DE DEPENDENCIAS  
  
Utiliza la inyección de dependencias para proporcionar implementaciones concretas de interfaces a través de constructores o métodos en lugar de crear esas implementaciones directamente en tu código. Esto facilita la sustitución de implementaciones y la prueba unitaria de componentes.**

**¿Cómo podría este principio simplificar la incorporación de nuevas funcionalidades en un proyecto de desarrollo de software en Java?**

**Facilita la expansión sin modificar código existente**

**Cuando se sigue el principio de DI, las nuevas funcionalidades pueden ser incorporadas con mínimos cambios en el código existente. En lugar de modificar clases y métodos que ya están funcionando, puedes implementar nuevas dependencias (por ejemplo, nuevos servicios, repositorios o módulos) y inyectarlas donde sea necesario.**

**¿De qué manera contribuye a la modularidad y mantenibilidad del código?**

**El código es más fácil de mantener, ya que la lógica está distribuida y no hay dependencias ocultas ni componentes que gestionan dependencias por sí mismos. Esto facilita la localización de problemas y la modificación de una parte del sistema sin afectar al resto. Además, si una parte del sistema se vuelve obsoleta, se puede reemplazar sin necesidad de refactorizar grandes secciones del código.**

**¿Cuál es el impacto de seguir este principio en la capacidad de realizar pruebas unitarias efectivas en el código Java?**

**Facilita la creación de mocks y stubs**

**La inyección de dependencias facilita enormemente la prueba de unidades aisladas. Dado que las clases no crean sus propias dependencias, puedes "mockear" (simular) fácilmente las dependencias utilizando bibliotecas de pruebas como Mockito. Esto te permite probar una clase sin necesidad de depender de implementaciones reales de servicios o componentes externos.**