Desarrollo web en entorno servidor

UT1.2 – Programación web

3 – Principios SOLID

IES Clara del Rey – Madrid

Principios SOLID

Cinco principios de diseño en programación orientada a objetos:

- **S** ingle Responsibility Principle (SRP)
- **O** pen-Closed Principle (OCP)
- **L** iskov Substitution Principle (LSP)
- **I** nterface Segregation Principle (ISP)
- **D** ependency Inversion Principle (DIP)

SOLID – Single Responsibility Principle

Enunciado (dos formas de expresarlo):

- Una clase (o un método) debe realizar una tarea, y sólo una.
- Una clase o método sólo debe tener una razón para cambiar.

Ventajas:

- Mejor mantenibilidad: Cada clase tiene una responsabilidad. Más fácil localizar y corregir errores sin afectar otras funcionalidades.
- Facilita la reutilización: Las clases que cumplen SRP suelen ser más reutilizables, ya que sólo hacen un trabajo específico.

SOLID – Single Responsibility Principle

Ventajas (continuación):

- Menor complejidad: Dividir el código en unidades más pequeñas y específicas lo hace más fácil de entender y manejar.
- Mayor flexibilidad para el cambio: Los cambios en una funcionalidad no afectarán a otras, reduciendo los efectos colaterales.
- Pruebas unitarias más sencillas: Es más fácil realizar pruebas unitarias en una clase, porque pruebas de diferentes responsabilidades no se mezclarán.

Ver ejemplos (correcto / incorrecto) en repositorio.

SOLID – Open Closed Principle

Enunciado:

- Open for extension, closed for modification.
- Abierto para la extensión, cerrado para la modificación.

Significa que una clase / interfaz / módulo:

- Estar abierta a la extensión: debe poder ampliarse sin necesidad de alterar su código fuente. Esto se logra con la herencia, los interfaces y el polimorfismo.
- Estar cerrada a la modificación: una vez que una clase/interfaz está terminada, probada y en uso, no debe modificarse. Si hay que añadir funcionalidad, debería extenderse (herencia), o hacer nuevas implementaciones.

SOLID – Open Closed Principle

Ventajas:

- Facilita la extensión: Permite añadir nuevas funcionalidades o comportamientos sin modificar el código existente.
- Evita la regresión: Al no modificar código existente para nuevas funcionalidades, se evita el riesgo de "romperlo".
- Mejor mantenibilidad: Se pueden añadir cambios sin cambiar código que ya funciona, reduciendo la complejidad en grandes sistemas.
- Código limpio: Ayuda a mantener la integridad y robustez del sistema a medida que crece.

Ver ejemplos (correcto / incorrecto) en repositorio.

SOLID – Open Closed Principle

Enunciado:

- Open for extension, closed for modification.
- Abierto para la extensión, cerrado para la modificación.

Significa que una clase / interfaz / módulo:

- Estar abierta a la extensión: debe poder ampliarse sin necesidad de alterar su código fuente. Esto se logra con la herencia, los interfaces y el polimorfismo.
- Estar cerrada a la modificación: una vez que una clase/interfaz está terminada, probada y en uso, no debe modificarse. Si hay que añadir funcionalidad, debería extenderse (herencia), o hacer nuevas implementaciones.

SOLID – Open Closed Principle

Ventajas:

- Facilita la extensión: Permite añadir nuevas funcionalidades o comportamientos sin modificar el código existente.
- Evita la regresión: Al no modificar código existente para nuevas funcionalidades, se evita el riesgo de "romperlo".
- Mejor mantenibilidad: Se pueden añadir cambios sin cambiar código que ya funciona, reduciendo la complejidad en grandes sistemas.
- Código limpio: Ayuda a mantener la integridad y robustez del sistema a medida que crece.

Ver ejemplos (correcto / incorrecto) en repositorio.

SOLID – Liskov Susbstitution Principle

Enunciado:

- Los objetos de una subclase deben poder sustituir a los objetos de la clase base sin alterar el comportamiento esperado del programa.

Significa que, si tenemos una clase base y una subclase:

- Se debe poder utilizar un objeto de la subclase en lugar de un objeto de la clase base, y el programa debería funcionar correctamente sin errores o comportamientos inesperados.
- Dicho de otra forma: Una subclase no debe romper las funcionalidades establecidas por la superclase, y debe respetar su contrato (es decir, las expectativas de comportamiento que establece la clase base).

SOLID – Liskov Susbstitution Principle

Ventajas:

- Más robustez: Al respetar LSP, las subclases no tendrán comportamientos inesperados, y el sistema será más estable.
- Facilita el uso de polimorfismo. El LSP es la base del porlimorfismo, las interfaces, y el código intercambiable.
- Más mantenibilidad: Al evitar sobrescribir comportamiento en subclases, es más fácil mantener y ampliar sin introducir errores.
- Previene errores en tiempo de ejecución. El polimorfismo hace que ciertos errores sólo se puedan detectar en tiempo de ejecución, y si cumplimos LSP, estos se minimizan.

Ver ejemplos (correcto / incorrecto) en repositorio.

SOLID – Interface Segregation Principle

Enunciado:

- Una clase no debe verse forzada a depender de interfaces que no necesita, ni a implementar sus métodos.
- De otra forma: es mejor tener muchos interfaces pequeños que uno grande que agrupe muchas funcionalidades, cuando muchas de ellas no se utilizarían

Si una interfaz es demasiado grande, con muchos métodos abstractos que realizan tareas diversas, una clase que necesite sólo una parte de estos métodos se verá forzada a implementar todos, aunque no los necesite.

Mejor dividir esta interfaz "grande" en varias más pequeñas. En cierta, medida, está relacionado con SRP.

SOLID – Interface Segregation Principle

Ventajas:

- Evita código innecesario: Dividir interfaces grandes en más pequeñas, evita implementar métodos que no se necesitan.
- Mejora comprensión y mantenimiento: Interfaces pequeñas son más fáciles de entender y mantener. Cada clase implementa lo que necesita.
- Mayor flexibilidad: Interfaces pequeñas permiten que las clases se adapten a cambios o nuevos requisitos más fácilmente.
- Previene el código "inflado": Interfaces grandes puede conducir a código que lanza excepciones o devuelve vacíos porque la clase realmente no necesita esos métodos. Ayuda a aplicar SRP.

Ver ejemplos (correcto / incorrecto) en repositorio.

SOLID – Dependency Inversion Principle

Enunciado:

- Las implementaciones (detalles) tienen que depender de las abstracciones, y no las abstracciones de las implementaciones.
- Los módulos de alto nivel deben depender de abstracciones de los módulos de bajo nivel, no de las implementaciones.

Explicación:

- Todos los módulos de un sistema deben depender de abstracciones (en Java interfaces), no de la implementación concreta de la abstracción.
- Esto permite que los detalles (implementaciones) cambien sin tener que modificar la abstracción, y sin tener impacto en los módulos que dependen de ella.

SOLID – Dependency Inversion Principle

Ventajas:

- Desacoplamiento: Al depender de abstracciones, no implementaciones, los módulos de alto nivel están menos acoplados a los de bajo nivel. El sistema será más flexible y fácil de cambiar o extender.
- Mantenibilidad: Como los detalles de bajo nivel pueden cambiar sin afectar al código de alto nivel, es más fácil de mantener a largo plazo.
- Facilidad para pruebas unitarias: Al utilizar abstracciones, es más sencillo sustituir las implementaciones reales por "mocks" o simulaciones, facilitando las pruebas unitarias.

SOLID – Dependency Inversion Principle

Ventajas (continuación):

- Sistemas más escalables y flexibles: Es más fácil añadir nuevas funcionalidades (nuevas implementaciones de bajo nivel) sin cambiar el comportamiento general del sistema.
- Facilita la inyección de dependencias: El DIP es fundamental para aplicar patrones como la inyección de dependencias (Dependency Injection), donde los módulos de bajo nivel se pasan como dependencias al módulo de alto nivel, generalmente en tiempo de ejecución. La inyección de dependencias es la forma de trabajo generalizada en la mayoría de frameworks de desarrollo, como Spring, por ejemplo.

Ver ejemplos (correcto / incorrecto) en repositorio.

SOLID – Consejos prácticos

- SRP:
  - Evitar clases, módulos o funciones muy grandes. Una clase muy grande es posible que hagas cosas de distintos ámbitos.
  - Analizar las clases, y buscar cosas que no encajen con la clase, cosas que podrían estar separadas, en otra clase, porque son algo distinto.
  - Específicamente, en el desarrollo web en servidor, intentar no mezclar lógica de negocio con presentación de los datos.
- OCP:
  - Pensar de antemano en cómo podríamos añadir nuevas funcionalidades sin modificar código existente.
  - Las interfaces son una buena opción de prepararse para el cambio.

SOLID – Consejos prácticos

- LSP:
  - Cuando heredes, evita hacerlo de forma que se rompa el comportamiento de la superclase.
  - Igualmente, al implementar interfaces, evita dejarlos a medio implementar, salvo que la especificación lo permita explícitamente (como en algunos métodos de Collection o List.
- ISP:
  - Evita interfaces "engordadas", con muchos métodos.
  - Mejor separa las interfaces en varias pequeñas, y hacer que las clases que necesiten más funcionalidad implementen más interfaces.

SOLID – Consejos prácticos

- DIP:
  - Cuando se trabaje con dependencias, que se dependa SIEMPRE de la abstracción, de la interfaz, nunca, salvo que sea imposible, de la implementación.
  - El punto anterior, intentar aplicarlo siempre que sea posible, se esté usando o no un contenedor IoC.