11ª Emisión

DIPLOMADO

Desarrollo de Sistemas con Tecnología Java

Módulo 2

Principios y Patrones de Diseño

Mtro. ISC Miguel Ángel Sánchez Hernández

DGTIC

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Dirección General de Cómputo y de Tecnologías de información y Comunicación
Dirección de Docencia en TIC

50 AÑOS Educación Continua 1971 - 2021

Objetivo

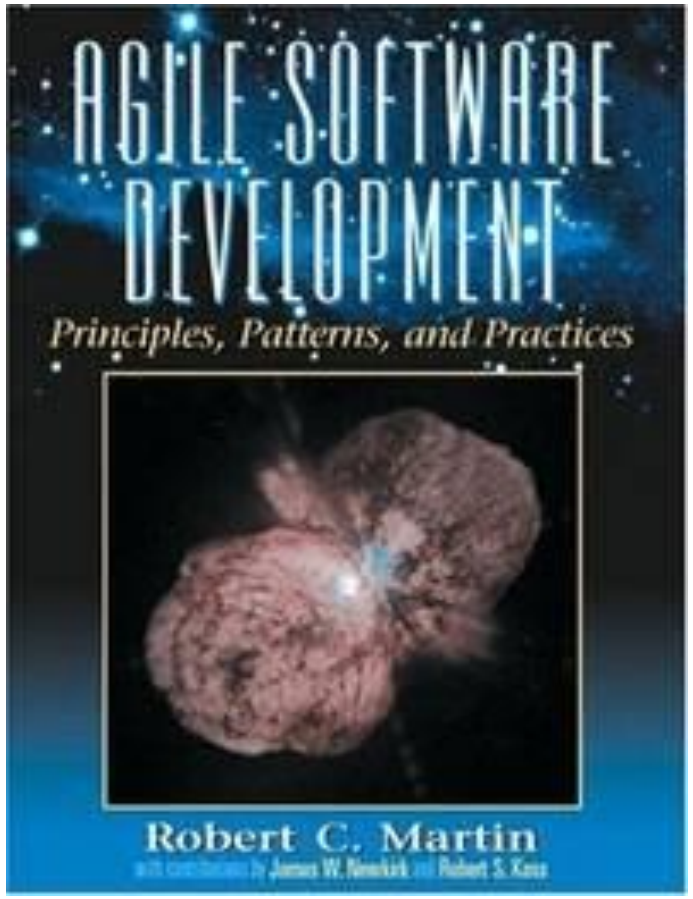
Conocer los principios SOLID

Lo que veremos

- Porque los principios SOLID son importantes
- Single Responsability Principle
- Open/Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Principios SOLID

- **Robert C. Martin (2002)**



Principios Básicos de la Programación Orientada a Objetos

- **S**ingle Responsability Principle
- **O**pen/Close Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsability Principal

En un diseño debemos tener una alta cohesión y un bajo acoplamiento.
Cohesión: Grado en que el contenido de una clase esta relacionada entre si.
Baja cohesión: Es una clase con funcionalidades muy dispersa.
Alta cohesión: Si una clase tiene una única responsabilidad.

“Preferible tener una alta cohesión”

Con esto tenemos **mutabilidad** y **reusabilidad**. Concluimos que una alta cohesión el código que tenemos esta especializado con una única tarea, es mas fácil de entender y mantener. El bajo acoplamiento la interacción ente las clases es mas rápido hacer una análisis y si relazamos un cambio no requerir actualizar muchas clases.

Single Responsability Principal

“Una clase debe tener una sola razón para cambiar.”

Decimos que debe tener una alta cohesión, clásico error de baja cohesión es combinar en un mismo clase lógica de negocio con la lógica de presentación.

Esto nos afecta porque si cambiamos la lógica de negocio, afecta a la lógica de presentación o al contrario si cambiamos la lógica de presentación se ve afectada la lógica de negocio.

Este principio nos dice que tenemos que crear dos clases diferentes, una para la lógica de negocio, y el otro para la lógica de presentación.

Práctica Single Responsability Principal

Se necesita hacer un aplicación donde se obtenga de la base de datos la información del alumno y sus materias, por parte del alumno se tienen los datos que son nombre y lista de materias, las materias se maneja el nombre de la materia y su calificación, lo que se quiere es que el sistema busque la calificación del alumno mayores o iguales a 5. También necesitamos la facilidad de exportar la información en CSV.

Open/Closed Principle

“¿Nuestro diseño puede estar preparado para que se extienda a futuro?”

La entidades que manejamos (clases, métodos), deben ser abiertas para la extensión, pero cerrada para la modificación.

La extensión que se quiere se debe conseguir sin modificar el código ya existente, porque si cambiamos el código existente al agregar una extensión, tendríamos que hacer pruebas de nuevo y posiblemente meteríamos errores a nuestro código.

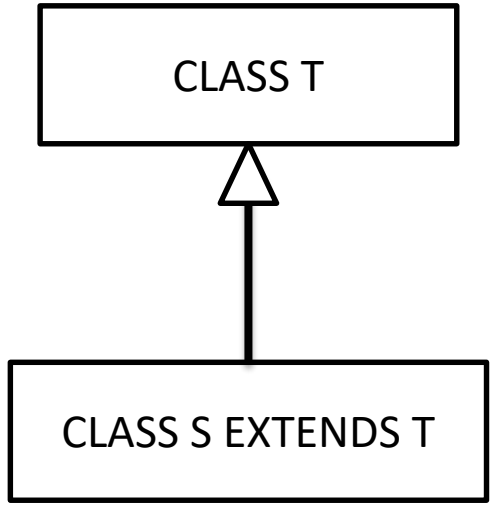
Práctica Open/Closed Principle

Se necesita realizar un módulo de un videojuego que permita pintar en la pantalla los elementos que son enemigos y personaje. Tanto enemigos y personajes deben tener la cualidad de poderlos dibujar en una posición que se indique.

LISKOV Substitution Principle

“¿En que condiciones la sustitubilidad de un tipo por otro alterar la corrección de un programa ?”

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.



Interface Segregation Principle

“Tener interfaces especificas para un tipo de cliente, que tengan una finalidad en concreto, es decir los clientes no deben ser forzados a tener métodos que ellos no usan”

Es mejor contar con muchas interfaces que tengan pocos métodos, que tener una sola y que se tenga que implementar todos los métodos que algunos no se usaran.

Dependency Inversion Principle

1. **Los módulos de alto nivel no deberían depender de módulos de bajo nivel, ambos deberían depender de abstracciones.**
2. **Las abstracciones no deberían depender de los detalles, los detalles deben depende de las abstracciones.**

¿Tengo que seguir firmemente a SOLID?

Podemos decir que son ambiguos, confusos, de complicar el código, de demorar el proceso de desarrollo. Pero Robert C. Martin en su artículo “Getting a SOLID start” **no se trata de reglas, ni leyes, ni verdades absolutas, sino más bien soluciones de sentido común a problemas comunes**. Son heurísticos, basados en la experiencia: “se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir.”

Lo que aprendimos

- Entender la filosofía de SOLID
- Aplicar ejemplos que demuestran como se puede interpretar a SOLID
- Comprender el camino de generar código escalable, mantenible y limpio