Curso de **Principios SOLID en C# y .NET**

Miguel Teheran

Prerrequisitos

Prerrequisitos

- Conocimiento en **C# y .NET**
- Conocimiento de **Git**
- **Visual Code** o **Visual Studio**
- **.NET 6** o superior

Buenas prácticas y código limpio

Buenas prácticas o Best practices

Buenas prácticas

- Resuelven desafíos de escenarios comunes
- Estándares comprobados y verificados
- Brindan guías fáciles de aprender y comprender
- Permiten tener una estructura similar para múltiples proyectos

Código limpio o Clean Code

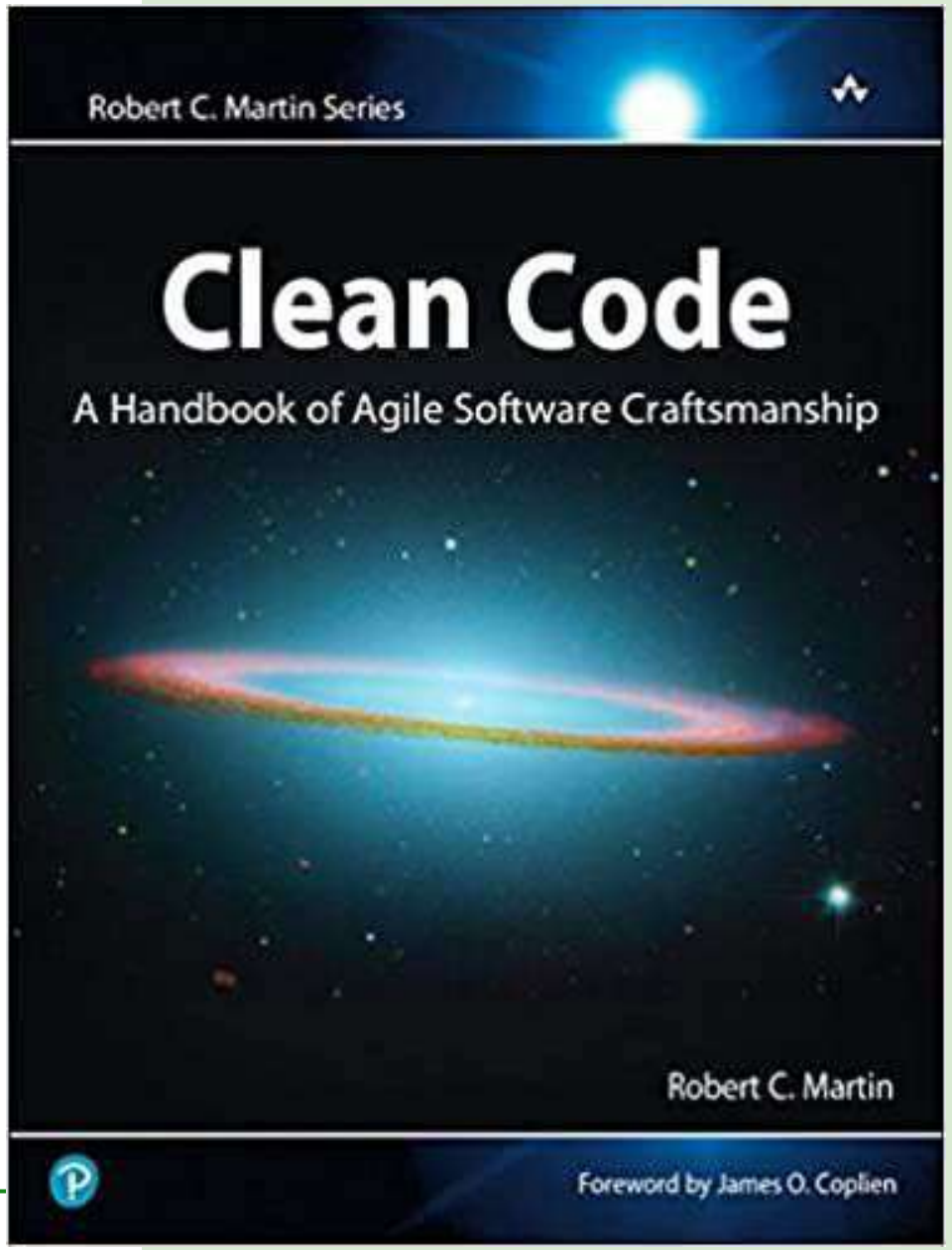
Código limpio

- Código fácil de entender y analizar
- Código fácil de mantener
- Código fácil de actualizar
- Código fácil de escalar

Cómo **logramos código limpio**

- Mantener bajo acoplamiento
- Utilizar sintaxis simple y actual
- Evitar incorporar muchas librerías de terceros
- Distribución de responsabilidades
- Creación de componentes pequeños

Libro recomendado



SOLID nos ayuda a
tener código limpio
y buenas prácticas

¿Qué son los **principios SOLID?**

S.O.L.I.D

S. Single responsibility principle - Principio de responsabilidad única

O. Open/closed principle - Principio de abierto/cerrado

L. Liskov substitution principle - Principio de sustitución de Liskov

S.O.L.I.D

I. Interface segregation principle - Principio de segregación de la interfaz

D. Dependency inversion principle - Principio de inversión de la dependencia

Características de **SOLID**

- Orientada al paradigma de orientación a objetos (POO)
- Se le atribuye a Robert C. Martin
- Diseño y refactorización de código
- Se debe implementar desde el inicio del proyecto cuando se crean los componentes

SOLID Ayuda a:

- Lograr código escalable
- Evitar deuda técnica (Technical debt)
- Plantear los fundamentos para el desarrollo guiado por pruebas (TDD)
- Tener un estándar claro en cada uno de los proyectos de un portafolio

Principio de **responsabilidad única**

Principio de **responsabilidad única**

- Single responsibility principle
- Distribuye las responsabilidades en diferentes componentes dentro de un sistema, donde cada componente tiene una única responsabilidad
- Aplica para módulos, clases, métodos y funciones

Ejemplo

Como usuario luego de confirmar la compra espero ver un mensaje de confirmación, tener la posibilidad de descargar la factura y un correo electrónico de confirmación.

Demo - **Escenario**

Tenemos una clase StudentRepository encargada de administrar los datos para el modelo student, pero en este momento no cumple con el principio de responsabilidad única, debe solucionarse.

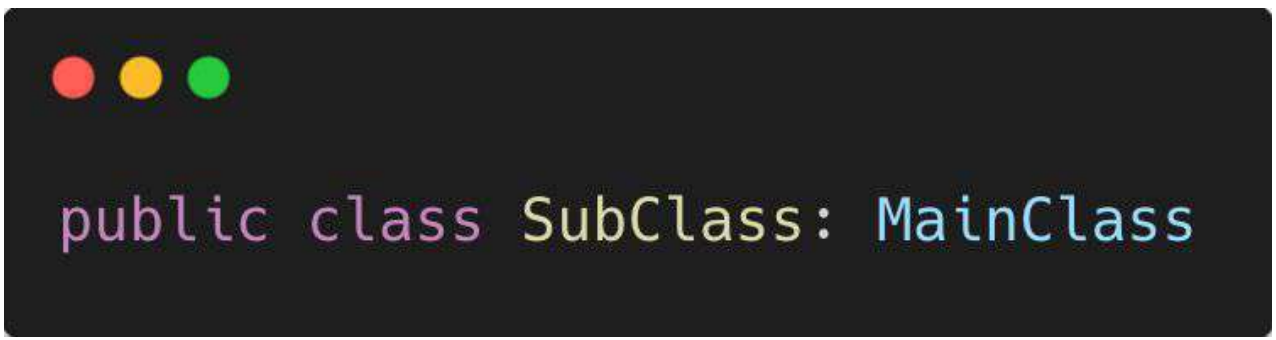
Principio de **abierto/cerrado**

Principio de **abierto/cerrado**

- Open/closed principle
- Bertrand Meyer - Robert Martin
- Un componente debe ser abierto para extensiones y cerrado para cambios
- Se deben utilizar tipos abstractos y múltiples implementaciones para lograrlo

El caso de C#

No soporta multi-herencia:



Soporta múltiples implementaciones de interfaces:

Demo - **Escenario**

Tenemos un proyecto que tiene una clase **EmployeeFullTime** para empleados a tiempo completo y **EmployeePartTime** para medio tiempo y necesitamos agregar **EmployeeContractor** para empleados externos cumpliendo con el principio Abierto/Cerrado.

Principio de **sustitución de Liskov**

Principio de **sustitución de Liskov**

- Liskov substitution principle
- Barbara Liskov - Robert Martin
- Tipo / Subtipo
- Los objetos dentro de un programa deben poderse reemplazar por subtipos de este sin alterar la lógica o el funcionamiento

Sea ϕ(x) una propiedad comprobable acerca de los objetos x de tipo T. **Entonces ϕ (y) debe ser verdad para los objetos y del tipo S, donde S es un subtipo de T.**

Demo - **Escenario**

En una aplicación tenemos 2 tipos de empleados, contractor y full-time, solo un empleado full-time tiene derecho a horas extras. El valor de la hora es 50 para full-time y de 40 para contractor. Ajustar el código para cumplir con los principios SOLID.

Principio de **segregación de la interfaz**

Principio de **segregación de la interfaz**

- Interface segregation principle
- Dividir interfaces grandes en pequeñas interfaces con una responsabilidad única
- Permite desacoplar los componentes y hacerlos crecer de una manera ordenada
- El principio de responsabilidad única aplicado a las interfaces

Demo - **Escenario**

Tenemos una interfaz que implementa múltiples actividades o responsabilidades de diferentes roles dentro de un proyecto, se debe aplicar el principio de ISP para distribuir de manera acorde las actividades e implementar solo lo necesario.

Principio de **inversión de la dependencia**

Principio de **inversión de la dependencia**

- Dependency inversion principle
- Busca desacoplar los componentes de la aplicación
- Nos ayuda a realizar cambios afectando el código lo menos posible
- Se debe implementar con tipos abstractos

Principio de **inversión de la dependencia**

- Existen 3 tipos de inyección de dependencia, por constructor, por propiedad y por parámetro
- Dependencia por constructor es la más común

La inyección de dependencias es fundamental para **poder aplicar la metodología Test driven development (TDD)**

Demo - **Escenario**

Tenemos una API que devuelve una lista de estudiantes. Esta API utiliza un repositorio que devuelve una colección de datos y utiliza una bitácora (logbook) para guardar los eventos o llamadas a la API. Debemos aplicar el principio de inversión de la dependencia en esta API.