ESCUELA POLITÉCNICA NACIONAL

CONSTRUCCIÓN Y EVOLUCIÓN DE SOFTWARE

NOMBRES ESTUDIANTES:	Mateo Calvache, Julián Camacho, Mateo Yunga
FECHA:	08-12-2025
TEMA:	**Clase 006 Taller Aplicando Principios de Código Limpio.**

1. Objetivo

Comprender los fundamentos teóricos del Código Limpio, analizar su importancia dentro del ciclo de desarrollo de software y reconocer cómo estos principios contribuyen a mejorar la mantenibilidad, escalabilidad y calidad del software en proyectos reales. Así mismo, identificar los elementos conceptuales necesarios para evaluar código y plantear mecanismos de mejora continua en bases de código colaborativas.

2. Introducción

El concepto de *Código Limpio* se refiere a la escritura de software que es claro, simple, legible y fácil de modificar. Según Robert C. Martin (autor de *Clean Code*), un código limpio "se lee como un texto bien escrito", presenta un flujo lógico sencillo y evita complejidad accidental. Esta idea también es reforzada en la clase del taller, donde se menciona que un código limpio se compone de elegancia, claridad y eficacia. [1]

La existencia de código limpio no solo facilita el trabajo individual, sino que impacta directamente el desempeño de los equipos, permitiendo aprender, mantener y extender el software sin generar deuda técnica innecesaria. [1] En entornos reales, donde los proyectos evolucionan constantemente, la calidad del código determina la velocidad con la que un equipo puede responder a cambios, corregir errores y agregar funcionalidades.

3. Importancia del Código Limpio en el Desarrollo de Software

3.1 Mantenibilidad

El código debe permitir modificaciones sin efectos secundarios inesperados. En el taller se indica que un código limpio reduce errores y facilita la integración de nuevos desarrolladores, ya que contiene menos complejidad innecesaria y una estructura más predecible [2]

Un proyecto con buena mantenibilidad requiere:

- Funciones pequeñas y enfocadas.
- Dependencias claras.

- Ausencia de duplicación.
- Lógica distribuida correctamente entre capas.

3.2 Escalabilidad

A medida que los sistemas crecen, también lo hace su base de código. Un software con buen diseño y principios de clean code ofrece:

- Modularidad.
- Claridad en responsabilidades.
- Facilidades para agregar nuevas características sin romper lo existente.

El taller menciona que esto "controla la complejidad y disminuye la deuda técnica" en proyectos en crecimiento.

3.3 Trabajo en equipo

Los proyectos modernos dependen del trabajo colaborativo. Sin código limpio, los miembros del equipo se enfrentan a interpretaciones ambiguas, errores frecuentes y retrasos.
Un código limpio permite:

- Entender rápidamente el propósito de cada módulo.
- Facilitar revisiones (code reviews).
- Mantener consistencia entre desarrolladores.
- Reducir curva de aprendizaje de nuevos integrantes.

4. Principios Fundamentales del Código Limpio

Basado en los conceptos explicados en el taller:

4.1 Nombres significativos

Los nombres deben comunicar intención de forma clara.
Ejemplo desde los repositorios analizados en el taller: métodos como `initCreationForm()` o `processFindForm()` expresan claramente su propósito .

Buenas prácticas:

- Evitar abreviaturas innecesarias.
- Usar verbos para funciones y sustantivos para objetos.
- Mantener coherencia en el vocabulario.

4.2 Funciones simples y con una sola responsabilidad (SRP)

Una función debe hacer solo una cosa, y hacerla bien.
En el taller se observa que ciertos métodos como `processFindForm()` tienen múltiples responsabilidades (validación, consulta y redirección), lo cual rompe dicho principio .

Ventajas:

- Facilita el testeo.
- Reduce errores.
- Mejora la legibilidad.

4.3 Comentarios útiles y no redundantes

Los comentarios no deben reemplazar un mal código, sino aclarar intenciones.
El taller recalca que “los comentarios deben aportar valor”, y no describir lo obvio o lo que podría explicarse con mejores nombres de variables.

4.4 Organización y estructura del código

Un buen software debe estar organizado de manera predecible, siguiendo patrones consistentes:

- Arquitecturas limpias (ej.: MVC).
- Separación clara entre capas.
- Módulos cohesivos.

En el taller se menciona que, aunque el código del proyecto elegido sigue MVC, existen oportunidades para separar responsabilidades que hoy residen innecesariamente en controladores .

4.5 Detección de olores de código (Code Smells)

Los “olores” no son errores, pero indican que algo debe mejorarse. Ejemplos señalados en el taller:

- Métodos demasiado largos.
- Lógica duplicada entre distintas funciones.
- Validaciones repetidas.
- Controladores excesivamente cargados.

Detectarlos permite prevenir deuda técnica y mejorar la calidad interna del software.

4.6 Pruebas y calidad

El código limpio debe ser:

- Fácil de probar.
- Diseñado para minimizar puntos de fallo.
- Compatible con pruebas unitarias y de integración.

El taller enfatiza que el refactor hecho bajo estos principios facilita generar pruebas que comprueben el comportamiento esperado del sistema.

Actividad Principal
Elige un repositorio público en GitHub

https://github.com/listerineh/pizza-planet-app.git

Archivos seleccionados del proyecto

Archivos elegidos:

1. order.py

2. report.py

https://github.com/listerineh/pizza-planet-app/blob/main/app/controllers/order.py

Análisis aplicando los principios de Código Limpio

order.py

```python
class OrderController(BaseController):
    manager = OrderManager
    __required_info = ('client_name', 'client_dni',
                       'client_address', 'client_phone', 'size_id')

    @staticmethod
    def calculate_order_price(size_price: float, ingredients: list, beverages: list):
        price = size_price + sum(ingredient.price for ingredient in ingredients) + \
            sum(beverage.price for beverage in beverages)
        return round(price, 2)

    @classmethod
    def create(cls, order: dict):
        current_order = order.copy()
        if not check_required_keys(cls.__required_info, current_order):
            return 'Invalid order payload', None

        size_id = current_order.get('size_id')
        size = SizeManager.get_by_id(size_id)

        if not size:
            return 'Invalid size for Order', None

        ingredient_ids = current_order.pop('ingredients', [])
        beverage_ids = current_order.pop('beverages', [])
        try:
            ingredients = IngredientManager.get_by_id_list(ingredient_ids)
            beverages = BeverageManager.get_by_id_list(beverage_ids)
            price = cls.calculate_order_price(
                size.get('price'), ingredients, beverages)
            order_with_price = {**current_order, 'total_price': price}
            return cls.manager.create(order_with_price, ingredients, beverages), None
        except (SQLAlchemyError, RuntimeError) as ex:
            return None, str(ex)
```

Ilustración 1 Order.py

¿Los nombres son claros?

Sí. En general, los nombres de clases, métodos y variables describen adecuadamente su propósito. OrderController comunica que gestiona órdenes; calculate_order_price expresa su función; create se entiende, aunque podría ser más explícito como create_order. Algunas mejoras posibles serían renombrar current_order a order_data. En general, los nombres son adecuados, pero con margen para ser más expresivos.

¿Las funciones son cortas y hacen una sola cosa?

Parcialmente. calculate_order_price sí cumple con SRP: solo calcula el precio. Sin embargo, create viola SRP al incluir validación, carga de entidades, cálculo de precio, preparación de datos, manejo de errores y construcción de la respuesta. Es un método multifuncional y debería dividirse.

¿Hay comentarios útiles o innecesarios?

No existen comentarios innecesarios, pero también faltan comentarios explicativos en decisiones críticas del flujo, como el uso de pop para ingredientes y bebidas o el manejo directo de excepciones SQLAlchemy.

¿Hay olores de código?

Sí, varios: método create demasiado grande, acoplamiento fuerte a capa de persistencia, manejo de errores mezclado con lógica de negocio, posible duplicación de validaciones y uso de tuples (result, error) que dificulta claridad.

¿Cómo está organizada la estructura?

El controlador actúa como coordinador, pero asume demasiada lógica de negocio. Está acoplado a managers, maneja excepciones, valida, arma payloads y construye respuestas. Carece de separación entre validación, negocio y persistencia.

Mejoras propuestas

1. Extraer validaciones obligatorias a un validador o servicio.
2. Dividir create en submétodos para reducir complejidad.
3. Crear un OrderService para encapsular lógica de negocio.
4. Manejar excepciones en la capa de servicio, no en el controlador.
5. Reemplazar respuestas basadas en tuples por un objeto o estructura clara.
6. Agregar comentarios útiles donde haya decisiones clave.

report.py

```python
from sqlalchemy.exc import SQLAlchemyError

from ..repositories.managers import ReportManager
from app.common.singleton import SingletonMeta

class ReportController(metaclass=SingletonMeta):
    manager = ReportManager

    @classmethod
    def get_report(cls):
        try:
            return cls.manager.get_report(), None
        except (SQLAlchemyError, RuntimeError) as ex:
            return None, str(ex)

    @classmethod
    def get_most_requested_ingredient(cls):
        try:
            return cls.manager.get_most_requested_ingredient(), None
        except (SQLAlchemyError, RuntimeError) as ex:
            return None, str(ex)

    @classmethod
    def get_more_revenue_month(cls):
        try:
            return cls.manager.get_more_revenue_month(), None
        except (SQLAlchemyError, RuntimeError) as ex:
            return None, str(ex)

    @classmethod
    def get_best_customers(cls):
        try:
            return cls.manager.get_best_customers(), None
        except (SQLAlchemyError, RuntimeError) as ex:
            return None, str(ex)
```

Ilustración 2 report.py

¿Los nombres son claros?

Sí. Los nombres de los métodos son descriptivos y comunican adecuadamente el propósito: get_report, get_most_requested_ingredient, get_more_revenue_month y get_best_customers. Sin embargo, existen ligeras inconsistencias de estilo, por ejemplo get_more_revenue_month podría ser más claro semánticamente. La propiedad manager también podría renombrarse a report_manager para mayor expresividad.

¿Las funciones son cortas y hacen una sola cosa?

Cada método realiza una única acción: llamar a un método del manager y manejar excepciones. Son métodos pequeños y cumplen parcialmente el principio de responsabilidad única. No obstante, existe duplicación del mismo patrón try/except en todos los métodos, lo que sugiere que podría abstraerse en un método auxiliar para evitar violar DRY.

¿Hay comentarios útiles o innecesarios?

No existen comentarios innecesarios, pero sí falta documentación en puntos clave como el uso de @classmethod, el propósito de SingletonMeta y la razón de retornar tuplas (resultado, error). Agregar comentarios ayudaría a otros desarrolladores a comprender el diseño.

¿Hay olores de código?

Sí, se identifican varios:
- Duplicación evidente del bloque try/except en cada método.
- Manejo de errores repetido en lugar de centralizado.
- Uso de SingletonMeta que puede generar acoplamiento y dificulta pruebas.
- Contrato de retorno basado en tuplas, lo cual puede generar ambigüedad.

¿Cómo está organizada la estructura?

La clase mantiene una estructura uniforme: métodos estáticos o de clase que actúan como fachada para ReportManager. La lectura es sencilla, pero la clase se vuelve demasiado fina y repetitiva, ya que no contiene lógica de negocio adicional y sólo replica llamadas al manager con manejo de errores.

Mejoras propuestas

1. Crear un método interno genérico para manejar llamadas seguras y eliminar duplicación.
2. Evaluar si SingletonMeta es realmente necesario.
3. Documentar la clase y los motivos del patrón utilizado.
4. Reemplazar tuplas por una estructura clara de respuesta.
5. Mejorar consistencia en los nombres y agregar tipado explícito para mayor claridad.

5. Conclusión

La teoría del Código Limpio constituye un pilar esencial en el desarrollo moderno de software, ya que promueve claridad, simplicidad y un diseño enfocado en la mantenibilidad. Tal como se revisa en el taller, incluso proyectos bien estructurados presentan oportunidades de mejora: duplicación de validaciones, funciones con demasiadas responsabilidades y controladores sobrecargados.

Esto demuestra que el código limpio no es un estado final, sino un proceso continuo**,** alineado con prácticas de refactorización constante y evolución de software. Aplicar estos principios permite construir sistemas más robustos, flexibles y fáciles de evolucionar, beneficiando tanto a desarrolladores individuales como a equipos completos.

6. Referencias

[1] R. C. Martin, *Código Limpio: Manual de estilo para el desarrollo ágil de software*, 2ª ed., Madrid: Anaya Multimedia, 2010.

[2] P. Natekar, "The Importance of Clean Code: Writing Maintainable Code for the Future", Medium, 2024. Disponible en: https://pawannatekar220.medium.com/the-importance-of-clean-code-27515760c2cc