Programación Avanzada

Sistema de Pedidos

Metodología del Desarrollo

Iteración - Modularidad - Validación - Escalabilidad

Desarrollo incremental:

- Permite entregar valor rápidamente con versiones funcionales desde el inicio.
- Facilita recibir retroalimentación temprana de los usuarios y mejorar en base a ella.
- Reduce riesgos al dividir el desarrollo en partes manejables.
- Se adapta mejor a cambios en los requisitos durante el proyecto.
- Favorece una mejor planificación y control del progreso.

```python


from envios.metodo_envio import MetodoEnvio

class Pedido:
    def __init__(self, items: list, peso_total_kg: float):
        self.items = items
        self.peso_total_kg = peso_total_kg
        self.metodo_envio: MetodoEnvio = None

    def asignar_metodo_envio(self, metodo: MetodoEnvio):
        self.metodo_envio = metodo

    def obtener_resumen_envio(self):
        if not self.metodo_envio:
            return "No se ha asignado un método de envío."
        costo = self.metodo_envio.calcular_costo(self.peso_total_kg)
        descripcion = self.metodo_envio.describir_envio()
        return (f"Resumen del Envío:\n"
                f"  Peso del paquete: {self.peso_total_kg} kg\n"
                f"  Método: {descripcion}\n"
                f"  Costo total del envío: ${costo:.2f}")
```

Metodología del Desarrollo

Iteración - **Modularidad** - Validación - Escalabilidad

Construcción por módulos funcionales:

- Facilita el desarrollo y mantenimiento al dividir el sistema en partes independientes.
- Permite que distintos equipos trabajen en paralelo sin interferencias.
- Aumenta la reutilización de código y mejora la organización del proyecto.
- Hace más simple detectar y corregir errores dentro de cada módulo.
- Mejora la escalabilidad y flexibilidad del sistema.

```python

from envios.metodo_envio import MetodoEnvio

class Pedido:
    def __init__(self, items: list, peso_total_kg: float):
        self.items = items
        self.peso_total_kg = peso_total_kg
        self.metodo_envio: MetodoEnvio = None

    def asignar_metodo_envio(self, metodo: MetodoEnvio):
        self.metodo_envio = metodo

    def obtener_resumen_envio(self):
        if not self.metodo_envio:
            return "No se ha asignado un método de envío."
        costo = self.metodo_envio.calcular_costo(self.peso_total_kg)
        descripcion = self.metodo_envio.describir_envio()
        return (f"Resumen del Envío:\n"
                f"  Peso del paquete: {self.peso_total_kg} kg\n"
                f"  Método: {descripcion}\n"
                f"  Costo total del envío: ${costo:.2f}")
```

Metodología del Desarrollo

Iteración - Modularidad - Validación - Escalabilidad

Validación y ajustes continuos:

- Nos permite detectar y corregir errores rápidamente.
- Asegura que el producto siempre cumpla con las necesidades del usuario.
- Mejora la calidad del software en cada etapa del desarrollo.
- Facilita adaptarse a cambios o mejoras sin rehacer todo el trabajo.
- Garantiza una evolución constante del sistema con menos riesgos.

```python

from envios.metodo_envio import MetodoEnvio

class Pedido:
    def __init__(self, items: list, peso_total_kg: float):
        self.items = items
        self.peso_total_kg = peso_total_kg
        self.metodo_envio: MetodoEnvio = None

    def asignar_metodo_envio(self, metodo: MetodoEnvio):
        self.metodo_envio = metodo

    def obtener_resumen_envio(self):
        if not self.metodo_envio:
            return "No se ha asignado un método de envío."
        costo = self.metodo_envio.calcular_costo(self.peso_total_kg)
        descripcion = self.metodo_envio.describir_envio()
        return (f"Resumen del Envío:\n"
                f"  Peso del paquete: {self.peso_total_kg} kg\n"
                f"  Método: {descripcion}\n"
                f"  Costo total del envío: ${costo:.2f}")
```

Metodología del Desarrollo

Iteración - Modularidad - Validación - **Escalabilidad**

Mejora progresiva de código:

- Permite optimizar el rendimiento y la calidad sin frenar el desarrollo.
- Facilita el refactoring continuo sin afectar la funcionalidad existente.
- Ayuda a mantener el código limpio, ordenado y fácil de escalar.
- Reduce la acumulación de deuda técnica a lo largo del proyecto.
- Mejora la comprensión y mantenibilidad del código por parte del equipo.

```python

from envios.metodo_envio import MetodoEnvio

class Pedido:
    def __init__(self, items: list, peso_total_kg: float):
        self.items = items
        self.peso_total_kg = peso_total_kg
        self.metodo_envio: MetodoEnvio = None

    def asignar_metodo_envio(self, metodo: MetodoEnvio):
        self.metodo_envio = metodo

    def obtener_resumen_envio(self):
        if not self.metodo_envio:
            return "No se ha asignado un método de envío."
        costo = self.metodo_envio.calcular_costo(self.peso_total_kg)
        descripcion = self.metodo_envio.describir_envio()
        return (f"Resumen del Envío:\n"
                f"  Peso del paquete: {self.peso_total_kg} kg\n"
                f"  Método: {descripcion}\n"
                f"  Costo total del envío: ${costo:.2f}")
```

Decisiones Técnicas

Modularización - Git - Decoradores

División de paquetes (Pedidos, Envios):

- Organiza el código según responsabilidades claras, facilitando su comprensión.
- Permite trabajar de forma modular, separando la lógica de pedidos y envíos.
- Mejora el mantenimiento y la escalabilidad del sistema.
- Facilita la reutilización y pruebas independientes de cada componente.
- Reduce errores al evitar mezclas innecesarias de funcionalidades.

```python

def aplicar_cupon(descuento: float):
    """
    Decorador que aplica un porcentaje de descuento al costo del envío.
    descuento: número entre 0 y 1 (por ejemplo, 0.10 para 10%)
    """
    def decorador(func):
        def wrapper(peso_kg):
            costo_original = func(peso_kg)
            costo_final = costo_original * (1 - descuento)
            return round(costo_final, 2)
        return wrapper
    return decorador
```

Decisiones Técnicas

Modularización - Git - Decoradores

Git para versionado y control de cambios:

- Permite llevar un historial claro y ordenado del desarrollo del proyecto.
- Facilita el trabajo en equipo con ramas y fusiones controladas.
- Ayuda a identificar, revertir y corregir errores de forma segura.
- Mejora la colaboración y seguimiento del progreso.
- Es una herramienta estándar, confiable y ampliamente utilizada.

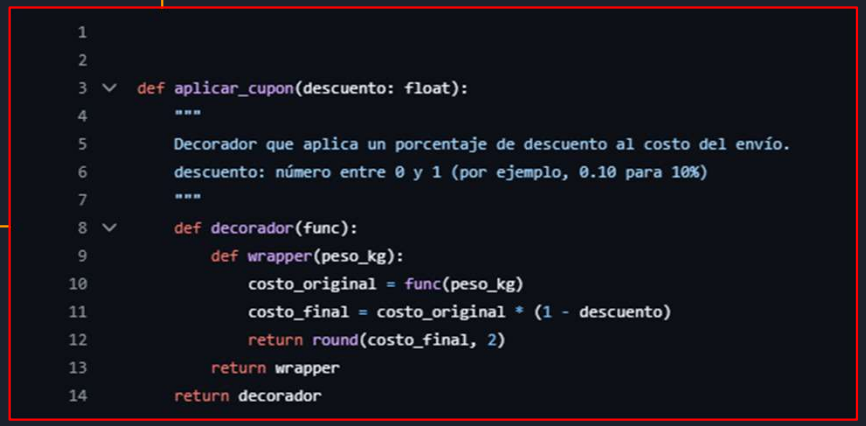
```python

def aplicar_cupon(descuento: float):
    """
    Decorador que aplica un porcentaje de descuento al costo del envío.
    descuento: número entre 0 y 1 (por ejemplo, 0.10 para 10%)
    """
    def decorador(func):
        def wrapper(peso_kg):
            costo_original = func(peso_kg)
            costo_final = costo_original * (1 - descuento)
            return round(costo_final, 2)
        return wrapper
    return decorador
```

Decisiones Técnicas

Modularización - Git - **Decoradores**

Decoradores para lógica dinámica:

- Permiten agregar funcionalidades sin modificar el código original.
- Favorecen un diseño más limpio, reutilizable y flexible.
- Facilitan aplicar comportamientos comunes (como validaciones o logs) en distintos módulos.
- Mejoran la separación de responsabilidades en el código.
- Ayudan a mantener la lógica central más simple y legible.



```python

def aplicar_cupon(descuento: float):
    """
    Decorador que aplica un porcentaje de descuento al costo del envío.
    descuento: número entre 0 y 1 (por ejemplo, 0.10 para 10%)
    """
    def decorador(func):
        def wrapper(peso_kg):
            costo_original = func(peso_kg)
            costo_final = costo_original * (1 - descuento)
            return round(costo_final, 2)
        return wrapper
    return decorador
```

Principios POO

Abstracción – Herencia – Polimorfismo – Encapsulamiento

Clases representativas del dominio:

- Representan entidades reales del problema.
- Hacen el código más fácil de entender.
- Facilitan cambios y mantenimiento.
- Organizan responsabilidades claras.
- Mejoran la reutilización y calidad del software.

```python

from .metodo_envio import MetodoEnvio

class EnvioPrioritario(MetodoEnvio):
    def __init__(self):
        super().__init__("Envío Prioritario")
        self.tarifa_base = 25.0
        self.costo_por_kg = 8.0

    def calcular_costo(self, peso_kg: float) -> float:
        return self.tarifa_base + (self.costo_por_kg * peso_kg)

    def describir_envio(self) -> str:
        return "Envío Prioritario: Entrega en 24 horas garantizada."
```

Principios POO

Abstracción – ***Herencia*** – Polimorfismo – Encapsulamiento

Reutilización y organización de código:

- Agrupan funcionalidades relacionadas en un solo lugar.
- Facilitan usar código ya creado en diferentes partes del proyecto.
- Mantienen el código ordenado y fácil de navegar.
- Evitar duplicación y errores repetidos.
- Mejoran la escalabilidad y mantenimiento del software.

```python

from .metodo_envio import MetodoEnvio

class EnvioPrioritario(MetodoEnvio):
    def __init__(self):
        super().__init__("Envío Prioritario")
        self.tarifa_base = 25.0
        self.costo_por_kg = 8.0

    def calcular_costo(self, peso_kg: float) -> float:
        return self.tarifa_base + (self.costo_por_kg * peso_kg)

    def describir_envio(self) -> str:
        return "Envío Prioritario: Entrega en 24 horas garantizada."
```

Principios POO

Abstracción – Herencia – **Polimorfismo** – Encapsulamiento

Intercambio dinámico de objetos:

- Permite cambiar objetos en tiempo de ejecución según la necesidad.
- Facilita adaptar el comportamiento del sistema sin modificar código.
- Mejora la flexibilidad y extensibilidad del proyecto.
- Reduce acoplamientos fuertes entre clases.
- Facilita pruebas y mantenimiento al poder reemplazar componentes fácilmente.

```python

from .metodo_envio import MetodoEnvio

class EnvioPrioritario(MetodoEnvio):
    def __init__(self):
        super().__init__("Envío Prioritario")
        self.tarifa_base = 25.0
        self.costo_por_kg = 8.0

    def calcular_costo(self, peso_kg: float) -> float:
        return self.tarifa_base + (self.costo_por_kg * peso_kg)

    def describir_envio(self) -> str:
        return "Envío Prioritario: Entrega en 24 horas garantizada."
```

Principios POO

Abstracción – Herencia – Polimorfismo – **Encapsulamiento**

Métodos públicos para manejar estado interno:

- Permiten controlar cómo se accede y modifica el estado de un objeto.
- Protegen los datos internos evitando cambios no deseados.
- Facilitan mantener la integridad y coherencia del objeto.
- Mejoran la encapsulación y modularidad del código.
- Permiten validar y controlar las operaciones sobre los datos.

```python

from .metodo_envio import MetodoEnvio

class EnvioPrioritario(MetodoEnvio):
    def __init__(self):
        super().__init__("Envío Prioritario")
        self.tarifa_base = 25.0
        self.costo_por_kg = 8.0

    def calcular_costo(self, peso_kg: float) -> float:
        return self.tarifa_base + (self.costo_por_kg * peso_kg)

    def describir_envio(self) -> str:
        return "Envío Prioritario: Entrega en 24 horas garantizada."
```

Estructura del Proyecto

Cohesión – Acoplamiento – Organización

Separación por responsabilidades:

- Cada módulo o clase tiene una función clara y específica.
- Facilita encontrar y modificar partes del código sin afectar otras.
- Mejora la mantenibilidad y escalabilidad del proyecto.
- Reduce dependencias innecesarias entre componentes.
- Promueve un código más limpio y organizado.

Estructura del Proyecto

Cohesión – **Acoplamiento** – Organización

Carpeta pedido: órdenes y productos:

- Contiene clases relacionadas con órdenes y productos.
- Centraliza todo lo que gestiona la compra y los items.

Carpeta envios: tipos y decoradores:

- Agrupa tipos de envío y decoradores para modificar su comportamiento.
- Permite extender o personalizar envíos sin alterar la lógica principal.

Patrones de Diseño

Estrategia - Decorador - Flexibilidad

Estrategia:

- Define familias de comportamientos intercambiables (por ejemplo, distintos tipos de envío).
- Permite cambiar el comportamiento en tiempo de ejecución sin modificar las clases cliente.

Decorador (Clase que agrega funcionalidad):

- Añade funcionalidades adicionales a un objeto sin alterar su estructura original.
- Facilita extender el comportamiento de forma flexible y reutilizable.

Decorador funcional con cupones:

- Aplica descuentos o beneficios extra agregando lógica sobre el envío o pedido.
- Permite combinar múltiples cupones o promociones sin modificar la base.

Resultados

Modularidad - Escalabilidad - Reutilización

- Sistema funcional y flexible, que se adapta a cambios.
- Base sólida para agregar nuevas funciones e integraciones.
- Código probado, claro y fácil de mantener.

Conclusión

Gracias a este proyecto pudimos profundizar en la exploración de conceptos avanzados de la programación orientada a objetos y pudiéndolos aplicar a un sistema funcional. El uso de la herencia, polimorfismo y el diseño modular junto con la implementación de decoradores nos permitió construir una solución flexible.