Trabajo Practico

Exploración de Conceptos Avanzados de POO en Python

Integrantes: Maximiliano Vergot

Juan Ignacio Marcos Merlo

Matias Daniel Alessandrello

Matias Santangelo

Universidad Nacional Guillermo Brown (UNAB)

Trabajo Practico

Índice

1.Introducción

El presente trabajo tiene como objetivo el desarrollo de un sistema orientado a objetos que simula el proceso de creación y gestión de pedidos con distintas modalidades de envío. El objetivo fue modelar de manera simple y ordenada las distintas partes que intervienen en un pedido, como los productos, el peso total y las diferentes modalidades de envío. A lo largo del desarrollo se aplicaron conceptos fundamentales de la POO como la herencia, el polimorfismo y la abstracción, lo que permitió construir un sistema flexible y extensible. También se incorporaron patrones de diseño como para organizar mejor el código y facilitar la incorporación de nuevas funcionalidades, como seguros, embalajes especiales o descuentos por cupones. El sistema fue pensado para funcionar desde la consola, permitiendo al usuario ingresar los datos de su pedido, elegir el tipo de envío y agregar servicios adicionales, todo de forma dinámica e intuitiva. Además, se priorizó una estructura modular y clara, separando la lógica en distintas carpetas según su responsabilidad, para favorecer el mantenimiento y la evolución del sistema en futuras versiones.

2.Metodología de Desarrollo

Si bien en el desarrollo de este proyecto no se adoptó formalmente una metodología específica, consideramos que la forma de trabajo implementada se alinea con los principios de la metodología incremental.
Esta estrategia permitió construir el sistema de manera progresiva, incorporando funcionalidades completas en cada iteración, lo que facilitó tanto la validación temprana de componentes como su integración gradual.
El enfoque incremental favoreció el desarrollo modular, permitiendo que cada parte del sistema fuese implementada y probada de forma independiente antes de su integración al conjunto.
A lo largo de las distintas iteraciones se realizaron ajustes continuos en la estructura del proyecto, incluyendo la organización de carpetas y la separación en módulos, lo que contribuyó significativamente a la escalabilidad, legibilidad y mantenibilidad del código.

3.Decisiones Técnicas

Estructura modular: Se optó por dividir el proyecto en múltiples paquetes (como pedido, envíos y decoradores) para favorecer el mantenimiento, la legibilidad y la escalabilidad del código.
Uso de decoradores funcionales: Para permitir la aplicación dinámica de descuentos a los envíos, se emplearon decoradores de funciones en Python, que modifican el comportamiento del método calcular_costo sin alterar su definición original.

Control de versiones: Se utilizó Git y GitHub para el versionado del proyecto, permitiendo llevar un historial detallado de los cambios, revertir errores y documentar avances mediante commits descriptivos.

4.Uso de la Programación Orientada a Objetos

El sistema fue desarrollado utilizando Programación Orientada a Objetos (POO), lo que permitió organizar el código de forma modular y mantener una estructura clara y escalable. Durante el desarrollo, se aplicaron varios de los principios fundamentales de la POO. Se hizo uso de la abstracción al identificar entidades clave como el pedido y los métodos de envío, representándolos como clases con atributos y comportamientos bien definidos. Esto permitió modelar el sistema de manera cercana a su funcionamiento real.
En cuanto al encapsulamiento, si bien no se utilizaron atributos privados de forma estricta (es decir, con doble guion bajo o atributos protegidos), las clases fueron diseñadas para que su estado interno se gestione mediante métodos públicos, promoviendo una separación entre la lógica interna y la interfaz que utiliza el resto del sistema. Es importante aclarar que se priorizó la simplicidad y la legibilidad del código, por lo que no se forzó el uso de encapsulamiento estricto, aunque se sentaron sus bases conceptuales.
El principio de herencia se aplicó especialmente en el diseño de las clases de envío, donde se definieron distintos tipos de envío a partir de una clase base común. Esto facilitó la reutilización de código y la extensión de comportamientos sin modificar el núcleo de la aplicación. Asimismo, el sistema aprovecha el polimorfismo, ya que todas las variantes de envío pueden utilizarse de forma intercambiable a través de métodos comunes. Este comportamiento se refuerza con el uso de decoradores funcionales que permiten modificar dinámicamente el cálculo del costo de envío, agregando funcionalidades como seguro, embalaje de regalo o descuentos por cupones, sin alterar la estructura original de los objetos.

5.Diseño y estructura

El sistema fue diseñado con un enfoque modular, empleando los principios de la Programación Orientada a Objetos (POO) para modelar las entidades clave del dominio. Esta aproximación facilitó una organización clara y coherente del código, asegurando alta cohesión dentro de cada módulo y un bajo acoplamiento entre ellos. La estructura del proyecto se dividió en carpetas que agrupan responsabilidades específicas, tales como pedido/ para la gestión de órdenes, envios/ para las diferentes modalidades y complementos de envío, y un archivo principal main.py que actúa como punto de entrada y proporciona la interfaz de interacción con el usuario.
Las clases principales utilizadas en el sistema son las siguientes:

Clase Pedido:

- Representa una orden compuesta por uno o varios productos. Esta clase encapsula la información relevante del pedido, como la lista de ítems y el peso total, y provee métodos

para asignar un método de envío específico y obtener un resumen detallado del mismo.

Clases de Envío (ubicadas en el paquete envíos/):

- EnvioEstandar
- EnvioExpress
- EnvioPrioritario

Estas clases implementan distintos algoritmos para calcular el costo de envío basado en el peso del paquete. Gracias al uso del polimorfismo, todas exponen una interfaz común mediante el método calcular_costo(peso), lo que permite que se puedan emplear de forma intercambiable en el sistema sin modificar la lógica que las consume.

Decoradores de Envío (también dentro del paquete envíos/):

- ConSeguroEnvio
- ConEmbalajeRegalo

Estas clases funcionan como decoradores que actúan como envoltorios (wrappers) sobre las instancias de los métodos de envío. Su propósito es extender dinámicamente el comportamiento original, añadiendo funcionalidades adicionales como seguro o embalaje especial sin alterar el código base de las clases de envío.

Patrones de diseño

En el diseño del sistema de pedidos y envíos se aplicaron dos patrones de diseño reconocidos: Strategy y Decorator. Estos patrones no fueron implementados de forma estricta siguiendo un esquema formal, pero están presentes en la arquitectura y la lógica del código, aportando flexibilidad y extensibilidad.
Para el patrón Strategy, se definió una clase abstracta que sirve como base para las diferentes modalidades de envío (estándar, exprés y prioritario). Sin embargo, aunque existe esta clase abstracta, el patrón no se aplica de manera explícita como un contexto que intercambia estrategias dinámicamente a través de una interfaz común formalizada, sino de forma más implícita. Cada modalidad concreta implementa su propio cálculo de costos, y el sistema permite seleccionar entre ellas, cumpliendo con la idea principal del patrón, aunque sin estructurarlo completamente como tal.
Por otro lado, el patrón Decorator fue utilizado de forma explícita para añadir comportamientos adicionales a los envíos sin alterar las clases originales. Las clases ConSeguroEnvio y ConEmbalajeRegalo son decoradores que reciben un objeto de tipo Envio y extienden su funcionalidad al calcular el costo total, agregando el importe del seguro o del embalaje respectivamente. Estos decoradores pueden aplicarse en cadena, lo que permite combinar múltiples características adicionales de manera dinámica y flexible.
Además, se implementó un decorador funcional en forma de función de orden superior (aplicar_cupon) que permite aplicar un porcentaje de descuento sobre el costo del envío. Este

enfoque demuestra que el patrón Decorator puede aplicarse tanto con clases como con funciones, aprovechando las capacidades del lenguaje Python.
La aplicación de estos patrones mejora significativamente la escalabilidad, el reuso de código y la mantenibilidad del sistema, permitiendo incorporar nuevas funcionalidades sin necesidad de modificar el núcleo del sistema.

6.Dificultades y resultados Obtenidos

A lo largo del desarrollo del proyecto enfrentamos diversas dificultades que nos llevaron a revisar decisiones y buscar soluciones en conjunto. Una de las principales complicaciones fue organizar correctamente la estructura del proyecto, especialmente en lo relacionado con los módulos y paquetes. Al principio tuvimos varios errores de importación al intentar conectar archivos ubicados en distintas carpetas, lo que nos obligó a investigar y entender mejor cómo funciona la carga de módulos en Python. También nos tomó tiempo lograr una implementación correcta de los decoradores, tanto en su forma con clases como en su versión funcional para aplicar descuentos. No fue sencillo modificar dinámicamente métodos ya existentes sin afectar el resto del comportamiento del sistema. Otro desafío importante fue lograr una buena separación de responsabilidades entre clases, manteniendo el sistema modular, pero asegurando que todas las partes pudieran interactuar correctamente. Finalmente, desarrollar una interfaz por consola que permita al usuario ingresar datos, elegir opciones y recibir un resumen claro también requirió varias pruebas y ajustes.

Luego del desarrollo del código obtuvimos los siguientes resultados:

- Construir un sistema funcional, capaz de simular pedidos con múltiples opciones de envíos y personalizaciones.

- La utilización de Programación orientada a objetos y patrones de diseño mejoró la claridad del código y la reutilización de este.

- El diseño modular nos permitió realizar pruebas independientes, mejorando y facilitando la detección y corrección de errores en etapas tempranas.

7.Conclusión

Gracias a este proyecto pudimos profundizar en la exploración de conceptos avanzados de la programación orientada a objetos, 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.