UNIVERSIDAD NACIONAL GUILLERMO BROWN

Trabajo Práctico Integrador

Calculadora de seguros

Sistema de cotización con Programación Orientada a Objetos

Integrantes: Sanchez Bajo Mauro (DNI: 22.864.483)
Sosa Redchuk Leandro Exequiel (DNI: 37.759.372)
Vera Espinola Julieta (DNI: 39.061.579)

Carrera: Licenciatura en Ciencia de Datos

Materia: Programación Avanzada (189)

Docente: Piriz Gianluca

Fecha: 20/06/2025

Índice

Introducción

Este informe detalla el desarrollo de una calculadora de seguros implementada en Python, cuyo objetivo fue diseñar un sistema flexible, extensible y con componentes autónomos, capaz de calcular cotizaciones de distintos tipos de seguros (auto y vida), con la posibilidad de aplicar descuentos u otras modificaciones dinámicamente.

Para lograrlo, se aplicaron los conceptos de Programación Orientada a Objetos (POO) abordados durante la cursada, en conjunto con la implementación de patrones de diseño ampliamente utilizados en el desarrollo de software.

La elección de este tema responde a la motivación de construir una solución práctica que no solo resuelva un problema concreto, sino que también demuestre cómo aplicar principios de diseño que favorezcan la reutilización, el bajo acoplamiento y la alta cohesión en los componentes del sistema.

Este proyecto tuvo como propósito consolidar los conocimientos adquiridos mediante la construcción de una aplicación funcional y mantenible, con una lógica de negocio realista y preparada para futuras ampliaciones. En particular, nos propusimos desarrollar una calculadora de seguros que:

- Calcule el costo de un seguro de auto y/o de vida;
- Permita diferentes tipos de seguros con reglas de cálculo específicas;
- Sea fácilmente extensible con nuevos tipos de seguros;
- Aplique descuentos, impuestos o recargos de forma flexible.

Metodología

1. Elección del tema

La elección del tema surgió a partir de la identificación de un problema real y cotidiano: la dificultad de elegir y cotizar un seguro adecuado entre múltiples opciones. Este escenario plantea una lógica de negocio concreta y variable según el tipo de seguro y la posibilidad de aplicar descuentos o recargos. Consideramos que este contexto representaba una excelente oportunidad para aplicar los conceptos aprendidos en Programación Orientada a Objetos (POO), debido a su necesidad de flexibilidad, extensibilidad y separación de responsabilidades.

2. Herramientas, lenguajes y tecnologías utilizadas

El proyecto fue desarrollado utilizando el lenguaje de programación Python, aprovechando su soporte para clases, herencia, encapsulamiento, interfaces abstractas (mediante abc) y polimorfismo.
Para el desarrollo y colaboración utilizamos la plataforma GitHub, lo que facilitó el control de versiones y el trabajo en equipo. Las pruebas y simulaciones del sistema se realizaron a través de la terminal y la consola, sin utilizar frameworks externos, ya que el foco estuvo puesto en la lógica de diseño y la estructura del código.

3. Diseño general del sistema

La arquitectura del sistema se basa en los principios de la Programación Orientada a Objetos (POO), y se apoya en la implementación de tres patrones de diseño fundamentales:

- Strategy: encapsula algoritmos de cálculo para distintos tipos de seguros.
- Decorator: permite aplicar descuentos o modificaciones al valor final de forma dinámica.
- Factory Method: se encarga de crear las estrategias de cálculo sin que el resto del código tenga que conocer los detalles de cómo se hacen.

Los componentes principales son:

- Estrategias de cálculo: definen cómo se calcula cada tipo de seguro (auto, vida).
- Seguro concreto: representa un seguro individual que utiliza una estrategia de cálculo.
- Decoradores: modifican dinámicamente el comportamiento de un seguro (por ejemplo, aplicando descuentos).
- Fábrica: crea las estrategias adecuadas sin acoplar el código cliente a implementaciones específicas.

4. Principios de Programación Orientada a Objetos

Durante el desarrollo del sistema se implementaron los cuatro pilares fundamentales de la Programación Orientada a Objetos:

- Abstracción: Se utilizaron clases abstractas (*EstrategiaSeguro*), definidas con la librería abc, para modelar comportamientos generales y forzar la implementación de

métodos en las subclases concretas. Esto permite trabajar a un nivel conceptual y ocultar detalles internos.

- Encapsulamiento: Cada clase contiene y gestiona su propio estado interno, protegiendo su lógica y evitando interferencias externas. Por ejemplo, el valor del seguro se maneja dentro de la clase y sólo es accesible mediante métodos definidos.
- Herencia: Las subclases (*SeguroAuto, SeguroVida, DescuentoPorEdad*, etc.) derivan propiedades y métodos de las superclases (*SeguroBase, DescuentoDecorator, EstrategiaSeguro*). Esto permite reutilizar código y definir comportamientos especificos segun el tipo de seguro.
- Polimorfismo: Gracias al uso de interfaces comunes (por ejemplo, todas las estrategias de cálculo implementan el método *calcular_costo()*), es posible intercambiar objetos y decoradores sin que el código cliente tenga que saber su tipo concreto.

Resultados

El sistema desarrollado cumple con los siguientes objetivos:

- Permite calcular el costo de diferentes tipos de seguros, como seguros de auto y de vida.
- Aplica reglas específicas de cálculo para cada tipo de seguro (por ejemplo, según la antigüedad del vehículo o la edad del cliente).
- Puede extenderse fácilmente con nuevos tipos de seguros sin modificar las clases existentes.
- Permite aplicar descuentos de manera flexible, gracias al uso de decoradores.
- Separa claramente la lógica de cálculo, la lógica de presentación y la creación de objetos, lo cual mejora la mantenibilidad y escalabilidad del sistema.

A modo de ejemplo, se destacan dos secciones clave del código implementado:

- Estrategia de cálculo para seguro de auto: Esta clase define cómo se calcula el seguro del auto, sumando una tarifa base y un recargo según la antigüedad del vehículo.

```python
class CalculoSeguroAuto(EstrategiaCalculo):
    def calcular(self, datos: dict) -> float:
        anio_actual = 2025
        tarifa_base = 10000
        antiguedad = anio_actual - datos.get("anio_auto", anio_actual)
        return tarifa_base + antiguedad * 500
```

- Decorador Descuento: este decorador aplica un 10% de descuento sobre el valor del seguro, sin alterar la clase original.

```python
class DescuentoClienteNuevo(SeguroBase):
    def __init__(self, seguro: SeguroBase):
        self.seguro = seguro

    def calcular(self) -> float:
        return self.seguro.calcular() * 0.9  # 10% de descuento
```

Discusión

El trabajo en grupo permitió distribuir tareas de manera eficiente y aprender de los aportes de cada integrante. El desarrollo del sistema generó espacios de discusión y toma de decisiones grupales, lo que favoreció el entendimiento conjunto del problema y de las posibles soluciones.

Una de las dificultades principales fue entender cómo aplicar correctamente los patrones de diseño. Si bien habíamos estudiado sus definiciones teóricas, llevarlos a la práctica requería entender cuándo y por qué usarlos. Esta dificultad se resolvió investigando ejemplos, revisando material de clase y probando distintas soluciones en conjunto.

Entre los principales aprendizajes, destacamos la importancia de:

- Diseñar el sistema pensando en el cambio futuro.
- Separar responsabilidades claramente en las clases.
- Utilizar patrones de diseño como herramientas reales, no solo teóricas.

Conclusiones

El desarrollo de esta calculadora de seguros nos permitió integrar de forma práctica los conocimientos adquiridos sobre Programación Orientada a Objetos y patrones de diseño, aplicándolos a una situación concreta y realista. Como resultado, logramos construir un sistema capaz de adaptarse a futuros cambios sin necesidad de modificar su estructura central.

Entre los logros alcanzados destacamos:

- Aplicación efectiva de los principios de herencia, abstracción, polimorfismo y encapsulamiento.

- Implementación de patrones de diseño (Strategy, Decorator y Factory Method) para mejorar la flexibilidad y mantenibilidad del código.
- Desarrollo de una solución práctica a un problema del mundo real, consolidando los conceptos aprendidos.

El diseño implementado demuestra varias cualidades clave:

- Desacoplamiento: cada clase cumple una única función, facilitando su comprensión y modificación.
- Extensibilidad: es posible incorporar fácilmente nuevos tipos de seguros o reglas de negocio sin alterar el código existente.
- Reutilización: las estrategias y decoradores pueden combinarse entre sí sin dependencia directa.

Como proyección a futuro, proponemos ampliar el sistema incluyendo:

- Nuevos tipos de seguros (hogar, salud, etc.).
- Registro de usuarios e integración con una base de datos.
- Generación de reportes en formato PDF o CSV.
- Conexión con plataformas de pago para simular un proceso completo de cotización y contratación.

En conclusión, el proyecto no solo fortaleció nuestros conocimientos técnicos, sino que también nos permitió experimentar el valor real de un buen diseño de software: aquel que no solo funciona, sino que está preparado para cambiar, crecer y mantenerse en el tiempo.