PATRONES Y PRINCIPIOS SOLID

SIMULADOR DEL CLIMA

HANNA KATHERINE ABRIL GÓNGORA
KAROL ASLEY ORJUELA MAPE

UNIVERSIDAD MANUELA BELTRÁN

PROGRAMACIÓN ORIENTADA A OBJECTOS

DOCENTE

DIANA MARCELA TOQUICA RODRÍGUEZ

BOGOTÁ DC LUNES 12 DE MAYO

1. **PREGUNTAS ORIENTADORAS**

1.2.¿Cómo se podría aplicar el principio de Responsabilidad Única en el diseño del Simulador de Clima para asegurar que cada clase tenga una única razón para cambiar?

- **Separación de responsabilidades:** Cada clase dentro del simulador debe tener una responsabilidad claramente definida y limitada. Por ejemplo, podría haber una clase para manejar la lógica del clima, otra para la representación gráfica, y otra para la interacción con el usuario.
- **Abstracción de funcionalidades:** Identifica las diferentes funcionalidades que necesita el simulador (por ejemplo, cálculo de la temperatura, simulación de precipitaciones, etc.) y agrúpalas en clases separadas. Cada clase debe ser responsable de una única funcionalidad.
- **Desacoplamiento de componentes:** Asegúrate de que las clases estén desacopladas entre sí tanto como sea posible. Esto significa que los cambios en una clase no deberían requerir cambios en otras clases, siempre que las interfaces entre las clases se mantengan estables.
- **Coherencia en la interfaz:** Las clases deben proporcionar una interfaz coherente y clara para interactuar con otras partes del sistema. Esto facilita la comprensión y el mantenimiento del código.
- **Evitar la sobrecarga de funcionalidades:** Evita que una clase acumule demasiadas responsabilidades. Si una clase está haciendo demasiadas cosas diferentes, considera dividirla en clases más pequeñas y especializadas.

1.3. ¿Qué estrategias se pueden utilizar para adherirse al principio de Abierto/Cerrado al diseñar el Simulador de Clima, permitiendo la extensión de funcionalidades sin modificar el código existente?

- **Uso de Interfaces y Abstracciones:**
  - Define interfaces para las funcionalidades clave del simulador, como el cálculo de temperatura, la simulación de precipitaciones, etc.
  - Las nuevas funcionalidades pueden implementar estas interfaces sin necesidad de modificar el código existente.

- **Patrones de Diseño:**
  - Utiliza patrones de diseño como el patrón de Estrategia o el patrón de Decorador.
  - El patrón de Estrategia permite encapsular algoritmos y cambiarlos dinámicamente, lo que facilita la adición de nuevas estrategias sin modificar el código existente.
  - El patrón de Decorador permite agregar comportamiento adicional a objetos existentes sin modificar su estructura.
- **Inyección de Dependencias:**
  - Utiliza la inyección de dependencias para proporcionar implementaciones concretas de funcionalidades a las clases principales del simulador.
  - Esto permite que las nuevas funcionalidades se agreguen como nuevas implementaciones de estas dependencias sin modificar el código existente.
- **Módulos o Plugins:**
  - Diseña el simulador de clima de manera que pueda cargar módulos o plugins de forma dinámica.
  - Los módulos pueden contener nuevas funcionalidades o algoritmos que se pueden agregar al simulador sin modificar el código principal.
- **Configuraciones Externas:**
  - Utiliza archivos de configuración externos para controlar el comportamiento del simulador.
  - Esto permite que nuevas funcionalidades se agreguen configurando el simulador para utilizar diferentes implementaciones de algoritmos o estrategias sin necesidad de modificar el código fuente.
- **Uso de Métodos y Clases Abstractas:**
  - Utiliza métodos abstractos en clases base que puedan ser implementados por subclases para proporcionar funcionalidades adicionales.
  - Las clases abstractas pueden definir comportamientos básicos, mientras que las subclases pueden extender o modificar estos comportamientos según sea necesario.

1.4 ¿Cómo se puede garantizar el cumplimiento del principio de Sustitución de Liskov al utilizar

herencia en el Simulador de Clima, asegurando que las subclases puedan ser usadas en lugar de las clases base sin alterar el comportamiento del programa?

- **Respetar las Invariantes de las Clases Base:** Las subclases deben respetar las invariantes definidas por las clases base. Esto significa que las condiciones que son verdaderas antes y después de la ejecución de un método en la clase base deben seguir siendo válidas en las subclases.
- **Preservar el Contrato de las Clases Base:** Las subclases deben respetar el contrato definido por las clases base. Esto incluye las precondiciones (condiciones que deben ser verdaderas antes de llamar a un método) y las postcondiciones (condiciones que deben ser verdaderas después de llamar a un método).
- **Extender, No Reemplazar:** Las subclases deben extender el comportamiento de las clases base sin modificarlo. Esto significa que pueden agregar funcionalidades adicionales o especializadas, pero no deben cambiar el comportamiento de los métodos heredados de manera que viole el contrato original.
- **No Introducir Excepciones No Esperadas:**
    - Las subclases no deben lanzar excepciones no esperadas o no documentadas que no sean compatibles con las clases base.
    - Si una clase base no lanza excepciones en determinadas situaciones, las subclases tampoco deberían hacerlo, a menos que sea absolutamente necesario y esté documentado adecuadamente.
- **Pruebas Rigurosas:**
    - Realiza pruebas rigurosas para garantizar que las subclases cumplan con el principio de Liskov.
    - Las pruebas deben cubrir todos los casos de uso y escenarios posibles, asegurándose de que las subclases puedan ser utilizadas en lugar de las clases base sin introducir errores o comportamientos inesperados.
- **Documentación Clara y Precisa:**
    - Documenta claramente el contrato y el comportamiento esperado de las clases base y sus subclases.
    - Esto ayudará a los desarrolladores que utilicen estas clases a comprender cómo

deben interactuar con ellas y qué esperar de su comportamiento.

2. **EXPLICACIÓN DEL CÓDIGO EN PYTHON (ANTES VS DESPUES)**

<table>
<tr><th>CODIGO ANTES DE IMPLEMENTACION SOLID</th><th>CODIGO DESPUES DE LA IMPLEMENTACION SOLID</th></tr>
<tr><td colspan="2">Parte lógica</td></tr>
<tr><td>

```python
from abc import ABC, abstractmethod
import requests

class Clima(ABC):
    def __init__(self, temperatura, humedad, presion):
        self.temperatura = temperatura
        self.humedad = humedad
        self.presion = presion

    @abstractmethod
    def obtener_condiciones(self):
        pass

```

</td><td>

```python
import tkinter as tk
from tkinter import messagebox
from abc import ABC, abstractmethod

#! PRINCIPAL MAS AVANZADO FUNCIONAL

class Clima(ABC):
    def __init__(self, temperatura, humedad, presion):
        self.temperatura = temperatura
        self.humedad = humedad
        self.presion = presion

    @abstractmethod
    def determinar_clima(self):
        pass
```

</td></tr>
<tr><td colspan="2">En la clase Clima, se define la estructura básica de los datos climáticos (temperatura, humedad, presion) y un método abstracto obtener_condiciones para obtener condiciones específicas del clima.

Esto cumple con SRP (Single Responsibility Principle) al tener una única responsabilidad: representar datos climáticos y proporcionar una interfaz para obtener condiciones climáticas específicas.

Interface Segregation Principle. Aunque no se define una interfaz explícita, se utilizan clases abstractas (ABC) y métodos abstractos para definir contratos claros. Cada clase cumple con su propia responsabilidad sin requerir métodos innecesarios.</td></tr>
</table>

```python
class CalculosClimaCalido(ABC):
    def __init__(self, clima: Clima):
        self.clima = clima

    @abstractmethod
    def determinar_clima(self):
        pass

    def soleado(self, temperatura, humedad, presion):
        return (
            temperatura ⩾ 30
            and temperatura ⩽ 35
            and humedad ⩾ 40
            and humedad ⩽ 45
            and presion ⩾ 1005
            and presion ⩽ 1010
        )
    def parcialmente_nublado(self, temperatura, humedad, presion):…
    def lluvioso(self, temperatura, humedad, presion):…
    def viento_fuerte(self, temperatura, humedad, presion):…
    def niebla(self, temperatura, humedad, presion):…
    def heladas(self, temperatura, humedad, presion):…
    def tormenta(self, temperatura, humedad, presion):…
    def calor_extremo(self, temperatura, humedad, presion):…
    def frio_extremo(self, temperatura, humedad, presion):…
```

```python
class CalculosClimaFrio(ABC):
    def __init__(self, clima: Clima):
        self.clima = clima

    @abstractmethod
    def determinar_clima(self):
        pass


    def soleado(self, temperatura, humedad, presion):
        return (
            temperatura ⩾ 20
            and temperatura ⩽ 25
            and humedad ⩾ 23
            and humedad ⩽ 27
            and presion ⩾ 1013
            and presion ⩽ 1017
        )
    def parcialmente_nublado(self, temperatura, humedad, presion):…
    def lluvioso(self, temperatura, humedad, presion):…
    def nieve(self, temperatura, humedad, presion):…
    def viento_fuerte(self, temperatura, humedad, presion):…
    def niebla(self, temperatura, humedad, presion):…
    def heladas(self, temperatura, humedad, presion):…
    def tormenta(self, temperatura, humedad, presion):…
    def calor_extremo(self, temperatura, humedad, presion):…
    def frio_extremo(self, temperatura, humedad, presion):…
```

```python
class CalculoClima(Clima):
    def determinar_clima(self):
        if (
            self.temperatura ⩾ 30
            and self.temperatura ⩽ 35
            and self.humedad ⩾ 23
            and self.humedad ⩽ 27
            and self.presion ⩾ 1013
            and self.presion ⩽ 1017
        ):
            return "Soleado"
```

Las clases CalculosClimaCalido y CalculosClimaFrio son extensiones de Clima y están diseñadas para ser extendidas pero no modificadas. Utilizan métodos abstractos para determinar diferentes tipos de climas basándose en condiciones específicas.

Esto cumple con OCP (Open/Closed Principle) al permitir la extensión de comportamientos de cálculo climático sin modificar las clases existentes.

```python
class CalculoClima(Clima, CalculosClimaCalido, CalculosClimaFrio):
    def __init__(self, ciudad, fecha, temperatura, humedad, presion):
        Clima.__init__(self, temperatura, humedad, presion)
        CalculosClimaCalido.__init__(self, self)
        CalculosClimaFrio.__init__(self, self)

        self.ciudad = ciudad
        self.fecha = fecha


    def determinar_clima(self):
        condiciones = self.obtener_condiciones()

        if self.temperatura ⩾25 :…


        else:…


    def obtener_condiciones(self):
        return (self.temperatura, self.humedad, self.presion)
```

- No hay una violación directa del principio de sustitución de Liskov. Las clases CalculosClimaCalido y CalculosClimaFrio pueden reemplazar a su superclase Clima sin cambiar el comportamiento esperado.
- Las subclases siguen el contrato de la superclase al implementar los métodos abstractos requeridos.
- La clase CalculoClima ejemplifica DIP (Dependency Inversion Principle) al depender de abstracciones (Clima, CalculosClimaCalido, CalculosClimaFrio) en lugar de implementaciones concretas. Esto permite una mayor flexibilidad y extensibilidad en el manejo de diferentes tipos de cálculos climáticos.

Parte grafica

```python
class VentanaPrincipal(tk.Tk):
    def __init__(self, clima_calculator):
        super().__init__()

        self.attributes("-fullscreen", True)
        self.bind("<Escape>", self.salir_pantalla_completa)
        self.clima_calculator = clima_calculator
        self.config(bg="white")
        self.crear_contenido()
        self.title("SkyScape")
        imagen_icono = Image.open("ui/icono_ventana.png")  # Cambia
        self.icon_photo = ImageTk.PhotoImage(imagen_icono)
        self.iconphoto(True, self.icon_photo)

        #* COLOCAR IMAGENES
        """self.imagen_tk = tk.PhotoImage(file="ui/icono_ventana.png

    def salir_pantalla_completa(self, event=None):
        self.attributes("-fullscreen", False)

    def crear_contenido(self):…

    def salir_programa(self):…

    def calcular_clima(self):…
```

```python
class SimuladorClimaApp(tk.Tk):
    def __init__(self):
        super().__init__()

        self.attributes("-fullscreen", True)
        self.bind("<Escape>", self.salir_pantalla_completa)
        self.crear_contenido()
        self.title("SkyScape")
        self.configure(bg="white")


    def salir_pantalla_completa(self, event=None):
        self.attributes("-fullscreen", False)

    def crear_contenido(self):…

    def salir_programa(self):
        self.destroy()
```

La clase VentanaPrincipal tiene la responsabilidad de representar la ventana principal de la aplicación GUI y manejar la interacción del usuario. Esto incluye la creación de widgets, la respuesta a eventos y la inicialización de la lógica de cálculo climático.

Cada método en VentanaPrincipal tiene una tarea específica relacionada con la interacción de la ventana principal.

La clase VentanaPrincipal recibe la lógica de cálculo climático (CalculoClima) como una dependencia, pero no está acoplada directamente a su implementación concreta. Esto permite una mayor flexibilidad y la posibilidad de cambiar la lógica de cálculo sin modificar la interfaz de usuario.

VentanaPrincipal depende de la abstracción CalculoClima, lo que facilita la sustitución de

diferentes implementaciones de cálculo climático.

```python
class FrameResultado(tk.Frame):
    def __init__(self, parent):
        super().__init__(
            parent,
            bg="grey",
            width=900,
            height=350,
            borderwidth=1,
            relief=tk.FLAT,
            highlightbackground="grey",
            highlightthickness=1,
        )
        self.parent = parent
        self.crear_widgets()

    def crear_widgets(self):
        pass
```

```python
    def frame_resultado(self):
        frame = tk.Frame(
            self,
            bg="grey",
            width=900,
            height=350,
            borderwidth=1,
            relief=tk.FLAT,
            highlightbackground="grey",
            highlightthickness=1,
        )
        frame.pack()

    def calcular_clima(self):
        try:
            temperatura = int(self.entrada_temperatura.get())
            humedad = int(self.entrada_humedad.get())
            presion = int(self.entrada_presion_at.get())

            clima = CalculoClima(temperatura, humedad, presion)
            tipo_clima = clima.determinar_clima()
            self.resultado_label.config(text=f"El clima es: {tip
        except ValueError:
            messagebox.showerror(
                "Error",
                "Ingrese valores numéricos válidos para temperat
            )

```

La estructura del programa permite la extensión sin modificar el código existente. Por ejemplo, podrías agregar más funcionalidades al FrameResultado sin necesidad de cambiar la implementación de otros componentes.

El FrameResultado está diseñado para ser extendido con nuevos widgets o funcionalidades sin alterar su funcionamiento básico.

```python
class FrameIngreso(tk.Frame):
    def __init__(self, parent):
        super().__init__(parent, bg="white")
        self.parent = parent
        self.crear_widgets()

    def crear_widgets(self):
        label_ciudad = tk.Label(
            self, text="Ciudad:", font=("Arial", 10, "bold"), bg="white"
        )
        label_ciudad.grid(row=0, column=0, padx=10, pady=5)
        self.entrada_ciudad = tk.Entry(self, width=15,relief=tk.FLAT,highlig
        self.entrada_ciudad.grid(row=0, column=1, padx=10, pady=5)

        label_fecha = tk.Label(
            self, text="Fecha (DD/MM/AAAA):", font=("Arial", 10, "bold"), bg
        )
        label_fecha.grid(row=1, column=0, padx=10, pady=5)
        self.entrada_fecha = tk.Entry(self, width=15,relief=tk.FLAT,highlig
        self.entrada_fecha.grid(row=1, column=1, padx=10, pady=5)

        label_temperatura = tk.Label(
            self, text="Temperatura (°C):", font=("Arial", 10, "bold"), bg="
        )
        label_temperatura.grid(row=2, column=0, padx=10, pady=5)
        self.entrada_temperatura = tk.Entry(self, width=15,relief=tk.FLAT,hi
        self.entrada_temperatura.grid(row=2, column=1, padx=10, pady=5)

        label_humedad = tk.Label(
            self, text="Humedad (%):", font=("Arial", 10, "bold"), bg="white
        )
        label_humedad.grid(row=3, column=0, padx=10, pady=5)
        self.entrada_humedad = tk.Entry(self, width=15,relief=tk.FLAT,highli
        self.entrada_humedad.grid(row=3, column=1, padx=10, pady=5)

        label_presion_at = tk.Label(
            self,
            text="Presión Atmosférica (hPa):",
            font=("Arial", 10, "bold"),
            bg="white",
        )
        label_presion_at.grid(row=4, column=0, padx=10, pady=5)
        self.entrada_presion_at = tk.Entry(self, width=15,relief=tk.FLAT,hig
        self.entrada_presion_at.grid(row=4, column=1, padx=10, pady=5)
```

```python
    def frame_ingreso(self):
        frame = tk.Frame(self, bg="white")
        frame.pack()

        label_temperatura = tk.Label(
            frame, text="Temperatura:", font=("Arial", 10, "bold")
        )
        label_temperatura.grid(row=4, column=0, padx=10, pady=5)
        self.entrada_temperatura = tk.Entry(frame, width=5)
        self.entrada_temperatura.grid(row=4, column=1, padx=10, pa
        unidades_temperatura = tk.Label(frame, text="°C", bg="whit
        unidades_temperatura.grid(row=4, column=2, pady=5)

        label_humedad = tk.Label(
            frame, text="Humedad:", font=("Arial", 10, "bold"), bg
        )
        label_humedad.grid(row=5, column=0, padx=10, pady=5)
        self.entrada_humedad = tk.Entry(frame, width=5)
        self.entrada_humedad.grid(row=5, column=1, padx=10, pady=5
        unidades_humedad = tk.Label(frame, text="%", bg="white")
        unidades_humedad.grid(row=5, column=2, pady=5)

        label_presion_at = tk.Label(
            frame,
            text="Presión Atmosférica:",
            font=("Arial", 10, "bold"),
            bg="white",
        )
        label_presion_at.grid(row=6, column=0, padx=10, pady=5)
        self.entrada_presion_at = tk.Entry(frame, width=5)
        self.entrada_presion_at.grid(row=6, column=1, padx=10, pad
        unidades_presion_at = tk.Label(frame, text="hPa", bg="whit
        unidades_presion_at.grid(row=6, column=2, pady=5)
```

No hay una violación directa del principio de sustitución de Liskov en este código. Las clases VentanaPrincipal, FrameIngreso y FrameResultado se comportan de acuerdo con sus propósitos y no introducen comportamientos inesperados. FrameIngreso funciona como se espera para un frame de ingreso de datos.

3. **RESUMEN CODIGO ANTERIOR VS CODIGO NUEVO, PRINCIPIOS SOLID**

PRINCIPIOS SOLID	**CODIGO ANTERIOR**	**CODIGO NUEVO LOGIC**	**CODIGO NUEVO INTERFACE**
S - Principio de Responsabilidad Única	Las clases Clima y CalculoClima cumplen con este principio, ya que Clima se encarga de definir los datos del clima y CalculoClima de calcular el clima basándose en esos datos.	Las clases tienen una única responsabilidad.	El diseño del código muestra una separación clara de responsabilidades.
O - Principio de Abierto/Cerrado	El código está abierto para extenderse con nuevas clases que implementen Clima o CalculoClima.	Se puede extender fácilmente creando nuevas clases de CalculosClimaCalido y CalculosClimaFrio.	La clase VentanaPrincipal está diseñada para ser extendida sin modificar su código original.
L - Principio de Sustitución de Liskov	No aplica directamente en este código.	Las clases de cálculo pueden ser intercambiadas.	No hay subclases explícitas en el código, por lo que no hay violaciones directas de LSP.

I - Principio de Segregación de Interfaces	Clima define una interfaz abstracta que las subclases deben implementar.	Las interfaces en las clases de cálculo definen métodos específicos.	El código muestra una segregación implícita de funcionalidades entre las clases.
D - Principio de Inversión de Dependencia	Las clases dependen de abstracciones (ABC) en lugar de implementaciones concretas.	Las clases de cálculo dependen de una abstracción (Clima).	La clase VentanaPrincipal no está acoplada directamente a una implementación específica de CalculoClima.

4. EXPLICACION DEL CODIGO UML CODIGO ACTUAL

➢ Simulador_clima_logic

```
@startuml simulador_logic

abstract class Clima {
    -temperatura: float
    -humedad: float
    -presion: float
    +obtener_condiciones(): void
}

interface CalculosClimaCalido {
    +soleado(temperatura: float, humedad: float, presion: float): bool
    +parcialmente_nublado(temperatura: float, humedad: float, presion: float): bool
    +lluvioso(temperatura: float, humedad: float, presion: float): bool
    +viento_fuerte(temperatura: float, humedad: float, presion: float): bool
    +niebla(temperatura: float, humedad: float, presion: float): bool
    +heladas(temperatura: float, humedad: float, presion: float): bool
    +tormenta(temperatura: float, humedad: float, presion: float): bool
    +calor_extremo(temperatura: float, humedad: float, presion: float): bool
    +frio_extremo(temperatura: float, humedad: float, presion: float): bool
    +determinar_clima(): void
}

interface CalculosClimaFrio {
    +soleado(temperatura: float, humedad: float, presion: float): bool
    +parcialmente_nublado(temperatura: float, humedad: float, presion: float): bool
    +lluvioso(temperatura: float, humedad: float, presion: float): bool
    +nieve(temperatura: float, humedad: float, presion: float): bool
    +viento_fuerte(temperatura: float, humedad: float, presion: float): bool
    +niebla(temperatura: float, humedad: float, presion: float): bool
    +heladas(temperatura: float, humedad: float, presion: float): bool
    +tormenta(temperatura: float, humedad: float, presion: float): bool
    +calor_extremo(temperatura: float, humedad: float, presion: float): bool
    +frio_extremo(temperatura: float, humedad: float, presion: float): bool
    +determinar_clima(): void
}

class ClimaTipico {
    -ciudad: str
    -fecha: str
    +obtener_clima_ciudad_fecha(): void
}

class CalculoClima {
    -ciudad: str
    -fecha: str
    -Clima
    -CalculosClimaCalido
    -CalculosClimaFrio
    +determinar_clima(): void
    +obtener_condiciones(): void
}

Clima <|-- CalculoClima
ClimaTipico o-- Clima
CalculoClima o-- CalculosClimaCalido
CalculoClima o-- CalculosClimaFrio

@enduml
```

- @startuml simulador_logic: Esto indica el inicio del diagrama UML y le da un nombre, en este caso, "simulador_logic".
- abstract class Clima { ... }: Aquí se define una clase abstracta llamada "Clima". Las clases abstractas en UML son aquellas que no pueden ser instanciadas

directamente y su propósito es proporcionar una estructura común para las clases derivadas. En este caso, la clase "Clima" tiene tres atributos privados (temperatura, humedad, presion) y un método público (obtener_condiciones()).

- interface CalculosClimaCalido { ... }: Se define una interfaz llamada "CalculosClimaCalido". Las interfaces en UML representan un conjunto de operaciones que una clase concreta debe implementar. En este caso, la interfaz contiene varios métodos que representan diferentes cálculos relacionados con el clima cálido.
- interface CalculosClimaFrio { ... }: Similar a la interfaz anterior, esta define un conjunto de métodos para cálculos relacionados con el clima frío.
- class ClimaTipico { ... }: Aquí se define una clase llamada "ClimaTipico". Esta clase tiene dos atributos privados (ciudad y fecha) y un método público (obtener_clima_ciudad_fecha()).
- class CalculoClima { ... }: Esta clase representa un cálculo específico del clima. Tiene varios atributos privados (ciudad, fecha, Clima, CalculosClimaCalido, CalculosClimaFrio) y dos métodos públicos (determinar_clima() y obtener_condiciones()).
- Clima <-- CalculoClima: Esta línea indica una relación de herencia entre las clases "Clima" y "CalculoClima". La flecha apunta desde la clase base (Clima) hacia la clase derivada (CalculoClima), lo que significa que "CalculoClima" hereda de "Clima".
- ClimaTipico o-- Clima: Esta línea representa una asociación entre las clases "ClimaTipico" y "Clima". El símbolo "o--" indica una relación de composición, lo que significa que un objeto de "ClimaTipico" contiene un objeto de "Clima".
- CalculoClima o-- CalculosClimaCalido: Similar a la asociación anterior, esta línea representa una relación de composición entre "CalculoClima" y "CalculosClimaCalido".
- CalculoClima o-- CalculosClimaFrio: También es una relación de composición entre "CalculoClima" y "CalculosClimaFrio".

- @enduml: Esta línea indica el final del diagrama UML.

➢ **Simulador_clima_ui**

```
@startuml  simulador_ui

class VentanaPrincipal {
    -clima_calculator: CalculoClima
    -icon_photo: ImageTk.PhotoImage
    +__init__(clima_calculator: CalculoClima)
    +salir_pantalla_completa(event: None)
    +crear_contenido()
    +salir_programa()
    +calcular_clima()
}

class FrameIngreso {
    -parent: VentanaPrincipal
    +__init__(parent: VentanaPrincipal)
    +crear_widgets()
}

class FrameResultado {
    -parent: VentanaPrincipal
    +__init__(parent: VentanaPrincipal)
    +crear_widgets()
}

class CalculoClima {
    +__init__(ciudad: str, fecha: str, temperatura: int, humedad: int, presion: int)
    +determinar_clima(): str
}

VentanaPrincipal "1"--* FrameIngreso
VentanaPrincipal "1"--* FrameResultado
VentanaPrincipal o-- CalculoClima

@enduml
```

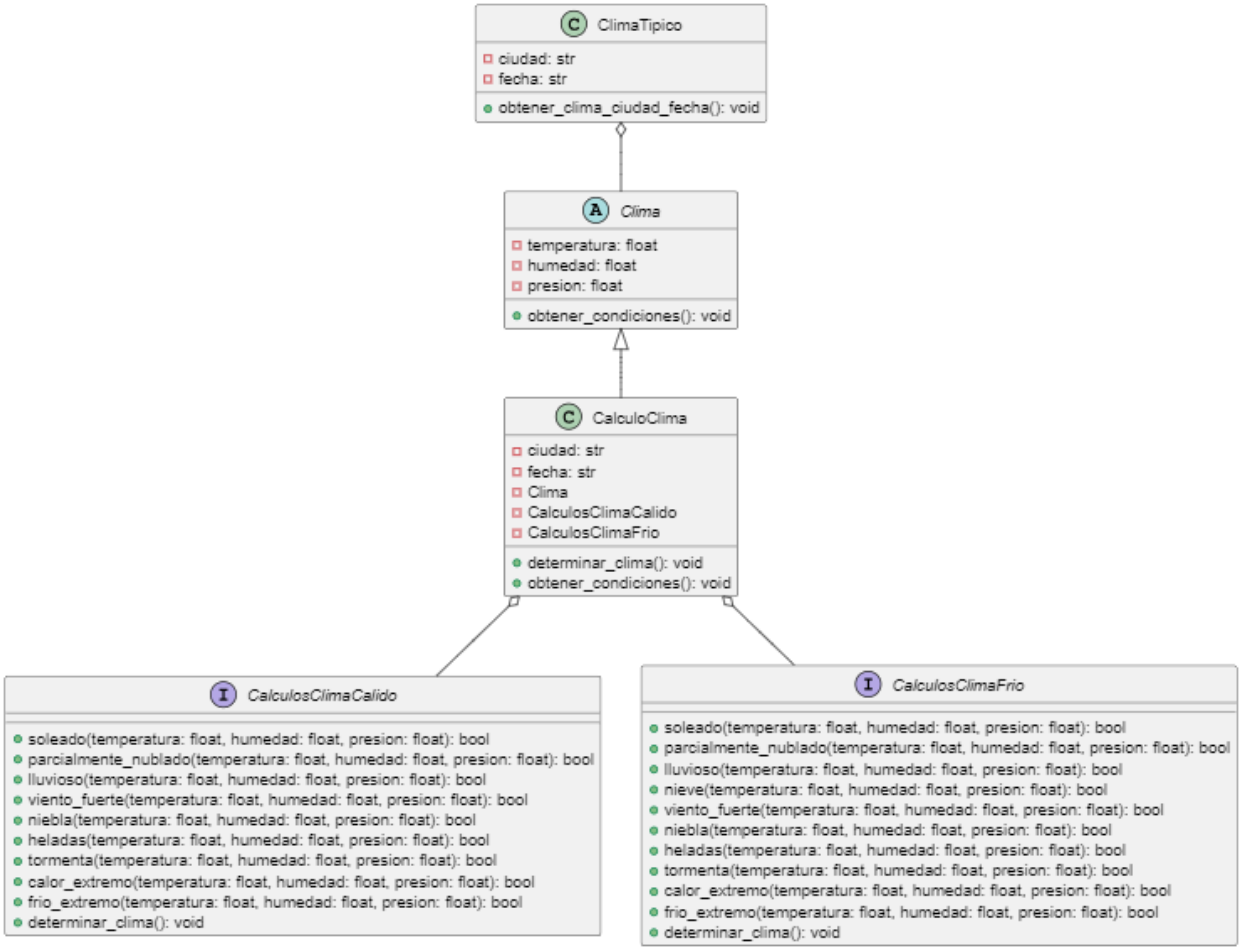
- @startuml simulador_ui: Esto indica el inicio del diagrama UML y le da un nombre, en este caso, "simulador_ui".

- class VentanaPrincipal { ... }: Se define una clase llamada "VentanaPrincipal", que representa la ventana principal de la interfaz de usuario. Esta clase tiene dos atributos privados (clima_calculator y icon_photo) y varios métodos públicos (__init__(), salir_pantalla_completa(), crear_contenido(), salir_programa(), calcular_clima()).

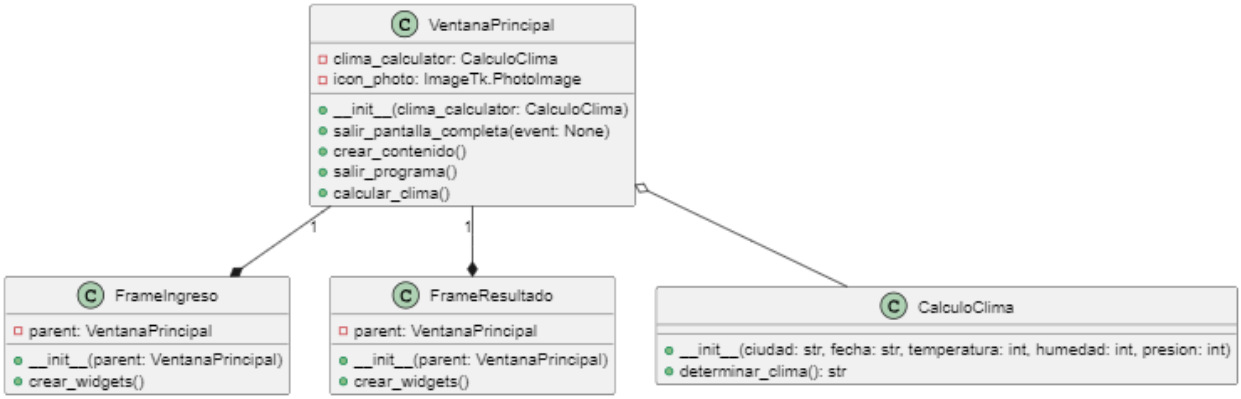
- class FrameIngreso { ... }: Aquí se define otra clase llamada "FrameIngreso", que representa un marco dentro de la ventana principal para ingresar datos relacionados con el clima. La clase tiene un atributo privado (parent) que representa la ventana principal y dos métodos públicos (__init__() y crear_widgets()).
- class FrameResultado { ... }: Similar a "FrameIngreso", esta clase representa otro marco dentro de la ventana principal para mostrar resultados relacionados con el clima. Tiene un atributo privado (parent) que representa la ventana principal y dos métodos públicos (__init__() y crear_widgets()).
- class CalculoClima { ... }: Esta clase representa el cálculo específico del clima. Tiene un constructor (__init__()) que toma varios parámetros relacionados con el clima y un método público (determinar_clima()) que devuelve una cadena de texto representando el tipo de clima.
- VentanaPrincipal "1"--* FrameIngreso: Esta línea indica una relación de agregación entre la clase "VentanaPrincipal" y "FrameIngreso". La cantidad "1" indica que una instancia de "VentanaPrincipal" puede tener múltiples instancias de "FrameIngreso".
- VentanaPrincipal "1"--* FrameResultado: Similar a la relación anterior, esta línea representa una relación de agregación entre "VentanaPrincipal" y "FrameResultado".
- VentanaPrincipal o-- CalculoClima: Aquí se muestra una relación de composición entre "VentanaPrincipal" y "CalculoClima". El símbolo "o--" indica que un objeto de "VentanaPrincipal" contiene un objeto de "CalculoClima".
- @enduml: Esta línea indica el final del diagrama UML.

5. **DIAGRAMA DEL CODIGO EN UML CODIGO ACTUAL**

- **Simulador_clima_logic**



- **Simulador_clima_ui**



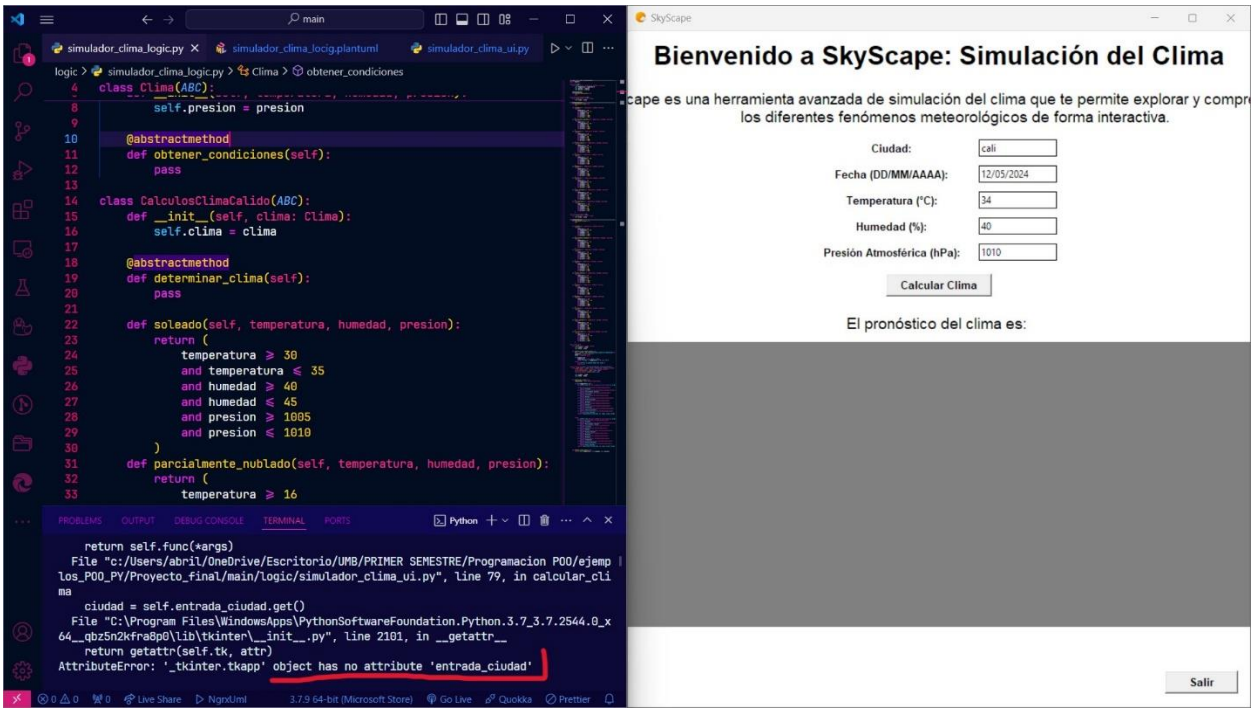
6. **EXPLICACION Y USO**

Este programa es un simulador de clima desarrollado en Python, con una estructura

orientada a objetos que sigue los principios SOLID y utiliza patrones de diseño. Las clases como Clima y Calculo Clima aplican el principio de responsabilidad única, donde Calculo Clima hereda y define condiciones para determinar el clima basado en datos ingresados por el usuario. La clase principal SimuladorClimaApp facilita la interacción con el usuario y el cálculo del clima correspondiente. Esta aplicación ilustra cómo la programación orientada a objetos, junto con los principios SOLID y los patrones de diseño, pueden ser empleados para crear herramientas interactivas que ayudan a comprender fenómenos meteorológicos

7. DEMOSTRACION DE PRUEBAS Y FUNCIONAMIENTO DEL CODIGO

Por motivos de modificaciones de la parte lógica y la parte gráfica, la funcionalidad aun esta en proceso, por lo que no se tienen pruebas de demostración de funcionamiento, pero a pesar de eso, el funcionamiento de la interfaz funciona junto con la modularización de la lógica, pero no da resultados correctos.



Como se muestra en la imagen, el código de la lógica trabaja junto al código de la parte

gráfica, pero aparece el error, subrayado en rojo, el cual esta en proceso de corrección.

8. **DECISIÓN DE DISEÑO O CONSIDERACION IMPORTANTE**

En este caso la decisión de diseño mas importante que se tomó fue haber implementado los principios de diseño SOLID para que la eficiencia del código fuera mejor, además se utilizaron estos principios porque promueven la creación de código más limpio, modular y mantenible. Al seguir estos principios, como la Responsabilidad Única, la Apertura/Cerrada, la Sustitución de Liskov, la Segregación de la Interfaz y la Inversión de Dependencias, se facilita la comprensión del código, se reducen los errores, se favorece la extensibilidad y se permite adaptarse a cambios futuros de manera más eficiente. Esto resulta en un desarrollo más efectivo y en la creación de sistemas informáticos más robustos y escalables.

9. **PILARES DE POO IMPLEMENTADOS**

Nuestra implementación busca abarcar los cuatro pilares fundamentales de la programación orientada a objetos. Aunque hemos priorizado la integración de estos conceptos en esta etapa inicial, nuestra estrategia contempla la inclusión progresiva de aspectos adicionales en etapas posteriores. Este enfoque nos permitirá construir un simulador climático sólido y adaptable, capaz de escalar y evolucionar con nuevas funcionalidades y mejoras de rendimiento. Esta estrategia se alinea con los principios SOLID de diseño de software. La aplicación de estos principios fomenta la extensibilidad y reutilización del código, fortaleciendo la cohesión y el bajo acoplamiento en nuestro diseño

10. **CONCLUCIONES**

El desarrollo de un simulador climático no solo implica avances técnicos, sino que también promueve habilidades creativas y de resolución de problemas. Este proyecto ofrece la oportunidad de aplicar conocimientos teóricos en un contexto práctico y significativo. La creación de una interfaz gráfica para interactuar con el sistema de

simulación no solo resulta en una herramienta funcional, sino que también explora nuevas formas de comunicar información compleja de manera clara y accesible. Este proyecto se alinea con los principios SOLID de diseño de software al fomentar el modularidad y la cohesión.

En resumen, esta iniciativa no solo permite aplicar conocimientos adquiridos, sino también profundizar en la comprensión de los principios y patrones de diseño de softwa