

PROYECTO FINAL
SIMULADOR DE CLIMA

HANNA KATHERINE ABRIL GÓNGORA
KAROL ASLEY ORJUELA MAPE

UNIVERSIDAD MANUELA BELTRÁN

PROGRAMACIÓN ORIENTADA A OBJETOS

DOCENTE

DIANA MARCELA TOQUICA RODRÍGUEZ

BOGOTÁ DC VIERNES 24 DE MAYO

1. PREGUNTAS ORIENTADORAS

1.1.¿Cómo se pueden modelar y simular diferentes condiciones climáticas utilizando POO?

La programación orientada a objetos (POO) es una forma de modelar y simular diferentes condiciones climáticas mediante la creación de clases y objetos que representen los elementos y comportamientos del clima. A continuación, se muestra un ejemplo de cómo se podría implementar esto utilizando POO:

- Crear una clase base llamada "Clima" que contenga atributos comunes para todas las condiciones climáticas, como temperatura, humedad, presión atmosférica, etc. Esta clase también puede tener métodos para obtener y establecer estos atributos.
- Crear clases derivadas para cada tipo específico de condición climática, como "Lluvia", "Nieve", "Sol", etc. Estas clases pueden heredar de la clase base "Clima" y agregar atributos y métodos adicionales según sea necesario.
- Utilizar polimorfismo para simular diferentes condiciones climáticas en función del tipo de objeto creado. Por ejemplo, se puede crear un objeto de la clase "Lluvia" con valores específicos para la intensidad

1.2.¿Qué roles juegan las clases abstractas y genéricas en la representación de los diversos elementos del clima?

Las clases abstractas y genéricas desempeñan un papel fundamental en la representación de los elementos del clima al proporcionar una estructura común y permitir la reutilización de código.

- Una clase abstracta es aquella que no puede ser instancia directamente, sino que sirve como plantilla para otras clases relacionadas. En el contexto del clima, se podrían definir clases abstractas para representar elementos generales como

"temperatura", "humedad" o "presión atmosférica". Estas clases abstractas podrían contener métodos comunes y propiedades básicas necesarias para todos los elementos del clima, como obtener el valor actual o realizar cálculos relacionados.

- Por otro lado, las clases genéricas son aquellas pueden trabajar con diferentes tipos de datos. En el caso del clima, se pueden utilizar clases genéricas para representar medidas específicas como "temperatura en grados Celsius" o "humedad relativa en porcentaje". Estas clases genéricas permiten adaptarse a diferentes tipos de datos sin necesidad de crear clase diferente para cada uno.

El uso de estas clases abstractas y genéricas permite una mayor flexibilidad y extensibilidad al modelar los diversos elementos del clima. Se pueden crear subclases concretas que hereden las propiedades definidas en las clases abstractas, que facilita la adición de nuevos elementos climáticos sin tener que reescribir todo el código desde cero.

1.3.¿Cómo se puede diseñar la interfaz de usuario para facilitar la interacción con el simulador y la visualización de los resultados?

- **Diseño intuitivo:** La interfaz debe ser fácil de entender y usar, incluso para usuarios sin experiencia previa en el uso de simuladores. Utiliza elementos familiares y organiza la información de manera lógica.
- **Navegación clara:** La estructura de la interfaz debe ser coherente y permitir a los usuarios moverse fácilmente entre diferentes secciones del simulador y los resultados.
- **Elementos visuales claros:** Utiliza colores, iconos y tipografías que ayuden a destacar la información importante y a guiar al usuario a través del proceso.
- **Instrucciones y ayuda contextual:** Proporciona instrucciones claras en cada paso del proceso y ofrece ayuda contextual cuando sea necesario. Esto puede ser en forma de mensajes emergentes, tutoriales o enlaces a documentación adicional.
- **Feedback inmediato:** Proporciona retroalimentación inmediata cuando el usuario interactúa con la interfaz, como confirmaciones de acciones realizadas o mensajes de error cuando se cometen errores.

- **Personalización:** Permite a los usuarios personalizar la interfaz según sus preferencias individuales, como la capacidad de ajustar el tamaño de los gráficos o seleccionar qué variables mostrar.
- **Interactividad:** Donde sea posible, permite a los usuarios interactuar directamente con los resultados, como mediante la manipulación de gráficos o la modificación de parámetros en tiempo real.
- **Compatibilidad multiplataforma:** Diseña la interfaz para que sea compatible con una variedad de dispositivos y tamaños de pantalla, incluidas computadoras de escritorio, tabletas y dispositivos móviles.
- **Documentación y tutoriales:** Proporciona acceso fácil a documentación detallada y tutoriales para ayudar a los usuarios a comprender mejor el funcionamiento del simulador y la interpretación de los resultados.
- **Pruebas de usabilidad:** Realiza pruebas de usabilidad con usuarios reales para identificar posibles problemas y realizar mejoras iterativas en el diseño de la interfaz.

1.4.¿De qué manera los principios SOLID pueden guiar el diseño de un sistema simulador robusto y mantenible?

Los principios SOLID son un conjunto de cinco principios de diseño de software orientado a objetos que promueven la creación de sistemas robustos, mantenibles y escalables. Aquí te muestro cómo cada uno de estos principios puede guiar el diseño de un sistema simulador:

- **Principio de Responsabilidad Única (SRP - Single Responsibility Principle):** Este principio establece que una clase debe tener una sola razón para cambiar. En el contexto de un simulador, esto significa que cada componente del sistema debe tener una única responsabilidad bien definida. Por ejemplo, podrías tener clases separadas para la lógica de simulación, la interfaz de usuario y la visualización de resultados. Esto facilita la comprensión del código y hace que sea más fácil de mantener y modificar en el futuro.
- **Principio de Abierto/Cerrado (OCP - Open/Closed Principle):** El principio OCP establece que las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación. En un simulador, esto significa que el código debe estar diseñado de manera que sea fácil agregar nuevas funcionalidades o modificar el comportamiento existente sin tener que cambiar el código base. Por ejemplo, podrías

diseñar el sistema usando patrones de diseño como el patrón de Estrategia, que permite intercambiar algoritmos de simulación sin modificar el código existente.

- **Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle):** Este principio establece que los objetos de un programa deben ser sustituibles por instancias de sus subtipos sin alterar la corrección del programa. En el contexto de un simulador, esto significa que las diferentes partes del sistema que interactúan entre sí deben poder ser intercambiadas por otras implementaciones sin afectar el funcionamiento del sistema en su conjunto. Por ejemplo, podrías tener diferentes tipos de simuladores que implementen la misma interfaz, lo que permite intercambiar fácilmente entre ellos.

- **Principio de Segregación de Interfaces (ISP - Interface Segregation Principle):** Este principio establece que una clase no debe depender de interfaces que no use. En el contexto de un simulador, esto significa que las interfaces deben ser lo más pequeñas y cohesivas posible, evitando la inclusión de métodos que no sean necesarios para todas las implementaciones. Esto ayuda a reducir el acoplamiento entre los diferentes componentes del sistema y facilita la creación de implementaciones alternativas sin afectar a otros componentes.

- **Principio de Inversión de Dependencias (DIP - Dependency Inversion Principle):** Este principio establece que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones. En el contexto de un simulador, esto significa que las dependencias entre los diferentes componentes del sistema deben ser gestionadas a través de interfaces o abstracciones en lugar de depender directamente de implementaciones concretas. Esto facilita la modificación y la extensión del sistema, ya que permite intercambiar fácilmente las implementaciones subyacentes sin afectar a los módulos de alto nivel.

2. EXPLICACIÓN DEL CÓDIGO EN PYTHON

En este caso cabe resaltar que el programa completo cuenta con 5 códigos, para hacer más fácil el entendimiento del código mediante modularidad y códigos SOLID.

2.1. Simulador_clima_logic.py

```
1 from abc import ABC, abstractmethod
2 import requests
```

- ABC y abstractmethod son clases de la biblioteca estándar abc que se utilizan para definir clases y métodos abstractos respectivamente.
- requests se usa para realizar solicitudes HTTP a una API externa.

```
4 class Clima(ABC):
5     def __init__(self, temperatura, humedad, presion):
6         self.temperatura = temperatura
7         self.humedad = humedad
8         self.presion = presion
9
10    @abstractmethod
11    def obtener_condiciones(self):
12        pass
13
14    class CalculosClimaCalido:
15        def __init__(self, clima: Clima):
16            self.clima = clima
17
```

- Clima es una clase abstracta que define propiedades comunes del clima como la temperatura, humedad y presión.
- @abstractmethod indica que obtener_condiciones es un método abstracto que debe ser implementado por las clases que heredan de Clima.

```

14 class CalculosClimaCalido:
15     def __init__(self, clima: Clima):
16         self.clima = clima
17
18     @abstractmethod
19 > def determinar_clima(self):...
21
22 > def soleado(self, temperatura, humedad, presion):...
31 > def parcialmente_nublado(self, temperatura, humedad, presion):...
40 > def lluvioso(self, temperatura, humedad, presion):...
49 > def viento_fuerte(self, temperatura, humedad, presion):...
58 > def niebla(self, temperatura, humedad, presion):...
67 > def heladas(self, temperatura, humedad, presion):...
76 > def tormenta(self, temperatura, humedad, presion):...
85 > def calor_extremo(self, temperatura, humedad, presion):...
94 > def frio_extremo(self, temperatura, humedad, presion):...
103
104 class CalculosClimaFrio:
105     def __init__(self, clima: Clima):
106         self.clima = clima
107
108     @abstractmethod
109 > def determinar_clima(self):...
111
112 > def soleado(self, temperatura, humedad, presion):...
121 > def parcialmente_nublado(self, temperatura, humedad, presion):...
130 > def lluvioso(self, temperatura, humedad, presion):...
139 > def nieve(self, temperatura, humedad, presion):...
148 > def viento_fuerte(self, temperatura, humedad, presion):...
157 > def niebla(self, temperatura, humedad, presion):...
166 > def heladas(self, temperatura, humedad, presion):...
175 > def tormenta(self, temperatura, humedad, presion):...
184 > def calor_extremo(self, temperatura, humedad, presion):...
193 > def frio_extremo(self, temperatura, humedad, presion):...
202

```

- Estas clases definen métodos para determinar diferentes tipos de condiciones climáticas dependiendo de si el clima es cálido o frío. Cada método toma como parámetros la temperatura, humedad y presión para evaluar las condiciones climáticas.

```

203 class ClimaTipico:
204     def __init__(self, ciudad, fecha):
205         self.ciudad = ciudad
206         self.fecha = fecha
207
208     def obtener_clima_ciudad_fecha(self):
209         url = f'https://api.openweathermap.org/data/2.5/weather?q={self.ciudad}&appid={self.api_key}'
210         respuesta = requests.get(url)
211         datos = respuesta.json()
212
213         if respuesta.ok:
214             temperatura = datos['main']['temp']
215             return "Cálido" if temperatura > 20 else "Frío"
216         else:
217             print("Error al obtener datos del clima.")
218             return None
219

```

- Esta clase se usa para obtener el clima típico de una ciudad en una fecha dada utilizando una llamada a la API de OpenWeatherMap.


```

229     def determinar_clima(self):
230         condiciones = self.obtener_condiciones()
231
232         if self.temperatura ≥ 25 :
233             print(f"El clima en {self.ciudad} el {self.fecha} es nor
234
235 >         if self.soleado(*condiciones):...
237 >         if self.parcialmente_nublado(*condiciones):...
239 >         if self.lluvioso(*condiciones):...
241 >         if self.nieve(*condiciones): # Este método no debería e
243 >         if self.viento_fuerte(*condiciones):...
245 >         if self.niebla(*condiciones):...
247 >         if self.heladas(*condiciones):...
249 >         if self.tormenta(*condiciones):...
251 >         if self.calor_extremo(*condiciones):...
253 >         if self.frio_extremo(*condiciones): # Este método no de
255             return "Condiciones no coinciden con ningún clima listad
256
257
258         else:
259             print(f"El clima en {self.ciudad} el {self.fecha} es nor
260 >         if self.soleado(*condiciones):...
262 >         if self.parcialmente_nublado(*condiciones):...
264 >         if self.lluvioso(*condiciones):...
266 >         if self.nieve(*condiciones):...
268 >         if self.viento_fuerte(*condiciones):...
270 >         if self.niebla(*condiciones):...
272 >         if self.heladas(*condiciones):...
274 >         if self.tormenta(*condiciones):...
276 >         if self.calor_extremo(*condiciones):...
278 >         if self.frio_extremo(*condiciones):...
280             return "Condiciones no coinciden con ningún clima listad
281

```

- CalculoClima es una clase que hereda de Clima, CalculosClimaCalido, y CalculosClimaFrio.
- Define un método determinar_clima para determinar el tipo de clima (cálido o frío) basado en las condiciones climáticas proporcionadas.
- También implementa obtener_condiciones para obtener las condiciones actuales de temperatura, humedad y presión.

2.2. Simulador_clima_ui.py

```
1  import tkinter as tk
2  from tkinter import messagebox
3  from PIL import Image, ImageTk
4  from frame_ingreso import FrameIngreso
5  from frame_resultado import FrameResultado
6  from simulador_clima_logic import CalculoClima
7
```

- Importamos módulos necesarios de tkinter para la interfaz gráfica, PIL para trabajar con imágenes y algunos otros archivos (FrameIngreso, FrameResultado, CalculoClima) relacionados con la lógica de simulación del clima.

```
10  class VentanaPrincipal(tk.Tk):
11 >     def __init__(self, clima_calculator):...
30
31
32
33     def salir_pantalla_completa(self, event=None):
34         self.attributes("-fullscreen", False)
35
36 >     def crear_contenido(self):...
85
86 >     def salir_programa(self):...
88
89 >     def calcular_clima(self):...
```

Inicialización de la ventana principal (__init__):

- Configuramos el tamaño y la posición de la ventana para que esté centrada en la pantalla.
- Configuramos el fondo, título y el icono de la ventana.

Métodos de la clase:

- salir_pantalla_completa: Este método permite salir del modo pantalla completa.
- crear_contenido: Aquí se crea el contenido de la ventana principal, incluyendo etiquetas, botones y el marco de ingreso de datos.
- salir_programa: Este método se llama cuando se presiona el botón "Salir" para cerrar la ventana y salir del programa.

- `calcular_clima`: Este método se ejecuta cuando se presiona el botón "Calcular Clima". Recopila los datos de entrada, realiza el cálculo del clima utilizando `CalculoClima` y muestra los resultados en una nueva ventana de resultado.

```
111 if __name__ == "__main__":
112     app = VentanaPrincipal(CalculoClima)
113     app.mainloop()
114
```

Aquí creamos una instancia de `VentanaPrincipal` y la ejecutamos con `mainloop()`, que inicia el bucle principal de la interfaz gráfica para que sea interactiva.

2.3. `Frame_ingreso.py`

```
1 import tkinter as tk
2
```

Aquí se importa la biblioteca `tkinter` bajo el alias `tk`, que se usa para crear interfaces gráficas en Python.

```
4 class FrameIngreso(tk.Frame):
5     def __init__(self, parent):
6         super().__init__(parent, bg="white")
7         self.parent = parent
8         self.crear_widgets()
9
```

Esta clase hereda de `tk.Frame`, lo que significa que es un marco (frame) dentro de una ventana `tkinter`. En el método `__init__`, se llama al constructor de la clase padre (`tk.Frame`) y se establece el color de fondo (`bg`) en blanco. También se guarda una referencia al widget padre (`parent`) y se llama al método `crear_widgets()` para crear los elementos de la interfaz gráfica.

```
10     def convertir_mayusculas(self, event):
11         texto = self.entrada_ciudad.get()
12         self.entrada_ciudad.delete(0, tk.END)
13         self.entrada_ciudad.insert(0, texto.upper())
14
```

Este método se utiliza para convertir el texto ingresado en el campo de entrada de la ciudad a mayúsculas. Se obtiene el texto actual con `self.entrada_ciudad.get()`, se borra el contenido del

campo con `self.entrada_ciudad.delete(0, tk.END)`, y luego se inserta el texto en mayúsculas con `self.entrada_ciudad.insert(0, texto.upper())`.

```
15     def crear_widgets(self):
16
17
18 >     label_ciudad = tk.Label(...)
21         label_ciudad.grid(row=0, column=0, padx=10, pady=5)
22     self.entrada_ciudad = tk.Entry(self, width=15, relief=tk.FLA
23     self.entrada_ciudad.grid(row=0, column=1, padx=10, pady=5)
24     self.entrada_ciudad.bind("<KeyRelease>", self.convertir_may
25
26 >     label_fecha = tk.Label(...)
29         label_fecha.grid(row=1, column=0, padx=10, pady=5)
30     self.entrada_fecha = tk.Entry(self, width=15, relief=tk.FLA
31     self.entrada_fecha.grid(row=1, column=1, padx=10, pady=5)
32
33 >     label_temperatura = tk.Label(...)
36         label_temperatura.grid(row=2, column=0, padx=10, pady=5)
37     self.entrada_temperatura = tk.Entry(self, width=15, relief=t
38     self.entrada_temperatura.grid(row=2, column=1, padx=10, pad
39
40 >     label_humedad = tk.Label(...)
43         label_humedad.grid(row=3, column=0, padx=10, pady=5)
44     self.entrada_humedad = tk.Entry(self, width=15, relief=tk.FL
45     self.entrada_humedad.grid(row=3, column=1, padx=10, pady=5)
46
47 >     label_presion_at = tk.Label(...)
50         label_presion_at.grid(row=4, column=0, padx=10, pady=5)
51     self.entrada_presion_at = tk.Entry(self, width=15, relief=tk
52     self.entrada_presion_at.grid(row=4, column=1, padx=10, pady
53
```

En este método se crean y configuran los diferentes widgets que formarán parte del marco de ingreso. Se crea un Label para etiquetar el campo de entrada de la ciudad, un Entry para que el usuario pueda ingresar la ciudad, y se configura la vinculación (bind) del evento `<KeyRelease>` al método `convertir_mayusculas`. Luego se repite el proceso para los campos de Fecha, Temperatura, Humedad y Presión Atmosférica.

2.4. Frame_resultado.py

```
1 import tkinter as tk
2 from image_manager import ImageManager
3 from PIL import Image, ImageTk
4
```

- tkinter: Es la biblioteca estándar de Python para crear interfaces gráficas de usuario (GUI).
- ImageManager, FrameIngreso: Son clases personalizadas importadas desde archivos externos.
- PIL.Image, ImageTk: Son módulos de la biblioteca Pillow (PIL) utilizados para trabajar con imágenes.

```

6  class FrameResultado(tk.Toplevel):
7      def __init__(self, parent, tipo_clima, ciudad, fecha, temperatura
8          super().__init__(parent)
9          self.parent = parent
10         self.tipo_clima = tipo_clima
11         self.ciudad = ciudad
12         self.fecha = fecha
13         self.temperatura = temperatura
14         self.humedad = humedad
15         self.presion = presion
16         self.config(bg="white")
17         self.title("Resultado del Clima")
18         width=1000
19         height=500
20         self.update_idletasks() # Asegurarse de que la ventana esta
21         screen_width = self.winfo_screenwidth()
22         screen_height = self.winfo_screenheight()
23         # Calcular la posición x e y para centrar la ventana
24         x = (screen_width // 2) - (width // 2)
25         y = (screen_height // 2) - (height // 2)
26         self.geometry(f"{width}x{height}+{x}+{y}")
27         self.title("SkyScape - Simulación del clima")
28         imagen_icono = Image.open("ui/icono_ventana.png")
29         self.icon_photo = ImageTk.PhotoImage(imagen_icono)
30         self.iconphoto(True, self.icon_photo)
31
32         self.image_manager = ImageManager(tipo_clima)
33         self.create_widgets()

```

- Se crea una nueva clase que hereda de tk.Toplevel, lo que significa que esta clase representa una ventana secundaria.
- `__init__`: Este método se ejecuta al crear una instancia de la clase. Recibe varios argumentos que representan datos sobre el clima.
- `super().__init__(parent)`: Llama al constructor de la clase base (tk.Toplevel) para inicializar la ventana.
- Se asignan los argumentos pasados al crear la instancia a variables de la clase.
- Se configura el fondo de la ventana como blanco y se establece el título como "Resultado del Clima".
- Se calcula la posición y tamaño de la ventana para centrarla en la pantalla.

- Se carga una imagen para el icono de la ventana y se convierte en un objeto `ImageTk.PhotoImage`.
- Se establece este objeto como el icono de la ventana.
- Se crea una instancia de `ImageManager` para manejar las imágenes relacionadas con el tipo de clima.
- Se llama al método `create_widgets` para generar los elementos de la interfaz gráfica.

```

35     def create_widgets(self):
36         image_label = self.image_manager.create_image_label(self)
37         if image_label:
38             image_label.pack(padx=60, pady=60)
39
40         resultado_label = tk.Label(
41             self,
42             text=f"El clima es: {self.tipo_clima}",
43             font=("Comic Sans MS", 16),
44             fg="#4361EE",
45             bg="white",
46         )
47         resultado_label.pack(padx=20, pady=20)
48
49         info_label = tk.Label(
50             self,
51             text=f"Ciudad: {self.ciudad}\n"
52                 f"Fecha: {self.fecha}\n"
53                 f"Temperatura: {self.temperatura}°C\n"
54                 f"Humedad: {self.humedad}%\n"
55                 f"Presión: {self.presion} hPa",
56             font=("Comic Sans MS", 14),
57             fg="#343a40",
58             bg="white",
59         )
60         info_label.pack(pady=10)
61
62

```

- En este método se crean los widgets que mostrarán la información de la ventana.
- Se crea un label para la imagen del clima (`image_label`) y otro para mostrar el tipo de clima (`resultado_label`).

```
63     def show(self):
64         self.transient(self.parent)
65         self.grab_set()
66         self.wait_window()
67
```

- Este método se utiliza para mostrar la ventana de resultado.
- `self.transient(self.parent)`: Hace que la ventana secundaria se comporte como hija de la ventana principal.
- `self.grab_set()`: Evita que se interactúe con otras ventanas mientras esta esté abierta.
- `self.wait_window()`: Espera hasta que se cierre la ventana antes de continuar con el código.

2.5. Image_manager.py

```
1  import tkinter as tk
2  from PIL import Image, ImageTk
3
```

- `tkinter`: Se importa para trabajar con interfaces gráficas.
- `PIL.Image, ImageTk`: Importaciones necesarias para trabajar con imágenes usando Pillow.


```

5  class ImageManager:
6      def __init__(self, tipo_clima):
7          self.tipo_clima = tipo_clima
8
9      def get_image(self, tipo_dato=None):
10         images = {
11             "Soleado": "ui/soleado.png",
12             "Parcialmente nublado": "ui/parcialmente_nublado.png",
13             "Lluvioso": "ui/lluvioso.png",
14             "Nevado": "ui/nevado.png",
15             "Vientos fuertes": "ui/viento_fuerte.png",
16             "Nublado": "ui/nublado.png",
17             "Heladas": "ui/heladas.png",
18             "Tormentas": "ui/tormenta.png",
19             "Calores extremos": "ui/temperaturas_extremas.png",
20             "Frios extremos": "ui/temperaturas_extremas.png",
21
22         }
23         return images.get(tipo_dato or self.tipo_clima, "")
24

```

- Se define una clase llamada ImageManager que se utilizará para gestionar imágenes.
- En el constructor `__init__`, se asigna el tipo de clima especificado al atributo `tipo_clima`.
- Este método devuelve la ruta de la imagen correspondiente al tipo de clima especificado. Si no se proporciona un tipo de clima específico, utiliza el tipo de clima asignado durante la inicialización.

```

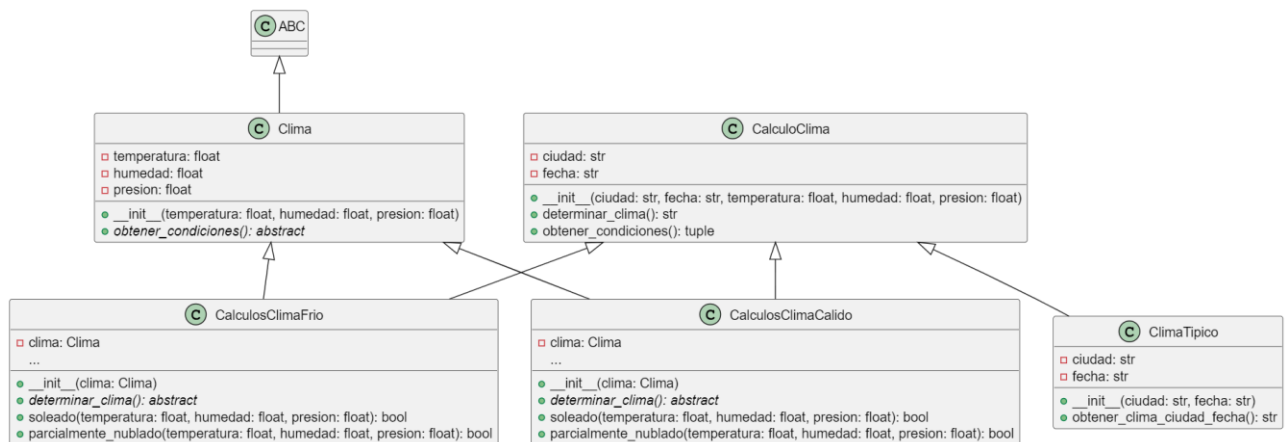
25     def create_image_label(self, parent, tipo_dato=None, width=None, height=None):
26         image_path = self.get_image(tipo_dato)
27         if image_path:
28             imagen_tk = tk.PhotoImage(file=image_path)
29             if width and height:
30                 imagen_tk = imagen_tk.subsample(width=width, height=height)
31             label_imagen = tk.Label(parent, image=imagen_tk)
32             label_imagen.image = imagen_tk
33             label_imagen.place(x=x, y=y)
34             return label_imagen
35         return None
36

```

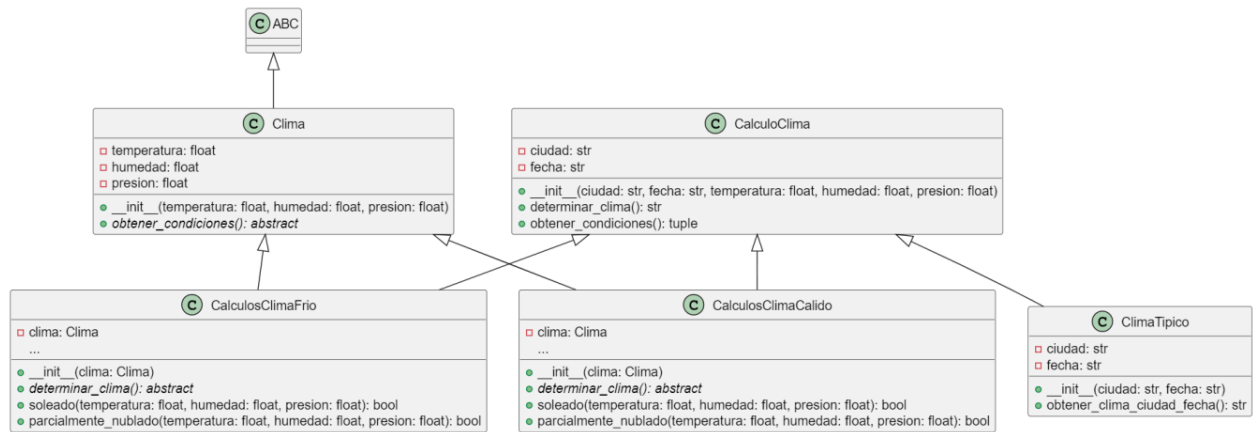
- Este método crea y devuelve un widget de etiqueta (tk.Label) que contiene una imagen. Toma el tipo de clima opcionalmente para determinar qué imagen cargar.
- `image_path = self.get_image(tipo_dato)`: Obtiene la ruta de la imagen utilizando el método `get_image`.
- `imagen_tk = tk.PhotoImage(file=image_path)`: Carga la imagen como un objeto `tk.PhotoImage`.
- `imagen_tk.subsample(width=width, height=height)`: Opcionalmente, reduce el tamaño de la imagen si se especifican `width` y `height`.
- `label_imagen = tk.Label(parent, image=imagen_tk)`: Crea un widget de etiqueta con la imagen.
- `label_imagen.image = imagen_tk`: Asigna la imagen al atributo `image` de la etiqueta para evitar que Python la elimine por la recolección de basura.
- `label_imagen.place(x=x, y=y)`: Coloca la etiqueta en la posición especificada en el padre.
- Devuelve la etiqueta creada o `None` si la imagen no se encontró.

3. DIAGRAMA DEL CODIGO EN UML

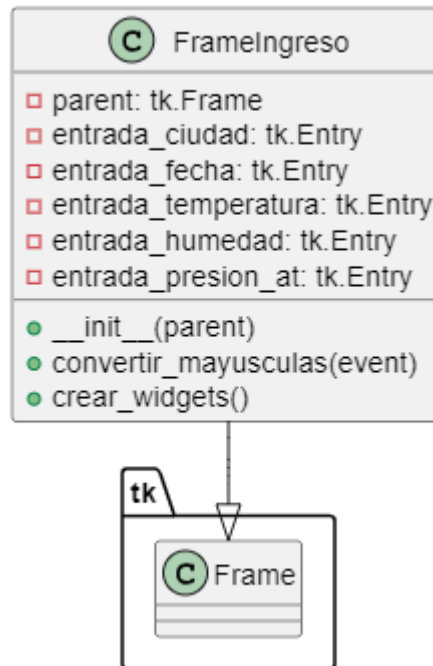
3.1 simulador_clima_logic



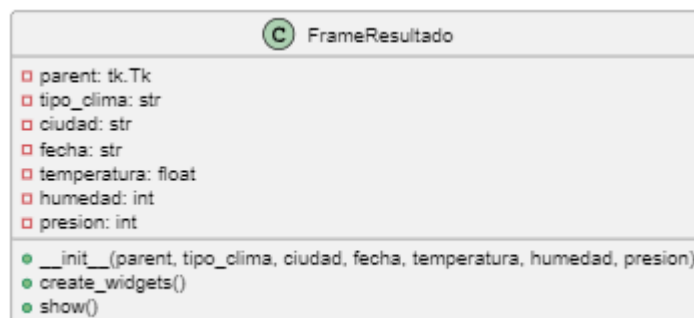
3.2 simulador_clima.ui




3.3 frame_ingreso



3.4 frame_resultado



3.5 image_manager

 ImageManager
□ tipo_clima: str
● <code>__init__(tipo_clima: str)</code>
● <code>get_image(tipo_dato: str): str</code>
● <code>create_image_label(parent: tk.Tk, tipo_dato: str, width: int, height: int, x: int, y: int): tk.Label</code>

4. EXPLICACION Y USO

Este es una aplicación de simulación del clima que combina lógica de cálculo, manejo de datos meteorológicos reales y una interfaz gráfica de usuario (GUI) para ofrecer una experiencia interactiva. Las clases `Clima`, `CalculosClimaCalido`, y `CalculosClimaFrio` definen modelos abstractos y concretos para representar diferentes tipos de clima y realizar cálculos basados en las condiciones meteorológicas proporcionadas. La clase `ClimaTipico` utiliza la API de OpenWeatherMap para obtener datos climáticos reales de una ciudad específica y determinar si el clima es cálido o frío. Por otro lado, la clase `CalculoClima` hereda de las clases de cálculo y de clima para realizar el análisis climático basado en las condiciones ingresadas por el usuario. La interfaz gráfica, implementada en los códigos de `VentanaPrincipal`, `FrameIngreso`, y `FrameResultado`, permite al usuario ingresar los datos necesarios (ciudad, fecha, temperatura, humedad, presión) y obtener un pronóstico del clima, mostrando resultados detallados y una imagen representativa del clima determinado. En conjunto, el programa ofrece una herramienta completa para explorar y comprender diversos fenómenos meteorológicos de manera interactiva y visualmente atractiva.

5. DEMOSTRACION DE PRUEBAS Y FUNCIONAMIENTO DEL CODIGO

5.1 Prueba 1. Ciudad típicamente cálida, fecha próxima

SkyScape

Bienvenido a SkyScape

Simulación del Clima

SkyScape es una herramienta avanzada de simulación del clima que te permite explorar y comprender los diferentes fenómenos meteorológicos de forma interactiva.

Ciudad:

LOS ANGELES

Fecha (DD/MM/AAAA):

24/05/2024

Temperatura (°C):

19

Humedad (%):

67


Presión Atmosférica (hPa):

990

Calcular Clima

Salir

SkyScape - Simulación del clima



El clima es: Vientos fuertes

Ciudad: LOS ANGELES

Fecha: 24/05/2024

Temperatura: 19°C

Humedad: 67%

Presión: 990 hPa

5.2 Prueba 2. Ciudad típicamente fría con fecha de invierno.

SkyScape

—

□

×

Bienvenido a SkyScape

Simulación del Clima

SkyScape es una herramienta avanzada de simulación del clima que te permite explorar y comprender los diferentes fenómenos meteorológicos de forma interactiva.

Ciudad:

OSLO

Fecha (DD/MM/AAAA):

12/13/2024

Temperatura (°C):

-7

Humedad (%):

44

Presión Atmosférica (hPa):

985

Calcular Clima

Salir

SkyScape - Simulación del clima

×



El clima es: Frios extremos

Ciudad: OSLO

Fecha: 12/13/2024

Temperatura: -7°C

Humedad: 44%

Presión: 985 hPa

5.3 Prueba 3. En caso de que los parámetros ingresados del clima no cumplan con ninguno.

SkyScape

Bienvenido a SkyScape

Simulación del Clima

SkyScape es una herramienta avanzada de simulación del clima que te permite explorar y comprender los diferentes fenómenos meteorológicos de forma interactiva.

Ciudad:

BOGOTA

Fecha (DD/MM/AAAA):

23/05/2024

Temperatura (°C):

13

Humedad (%):

65

Presión Atmosférica (hPa):

1012

Calcular Clima

Salir

SkyScape - Simulación del clima

El clima es: Condiciones no coinciden con ningún clima listado aquí

Ciudad: BOGOTA

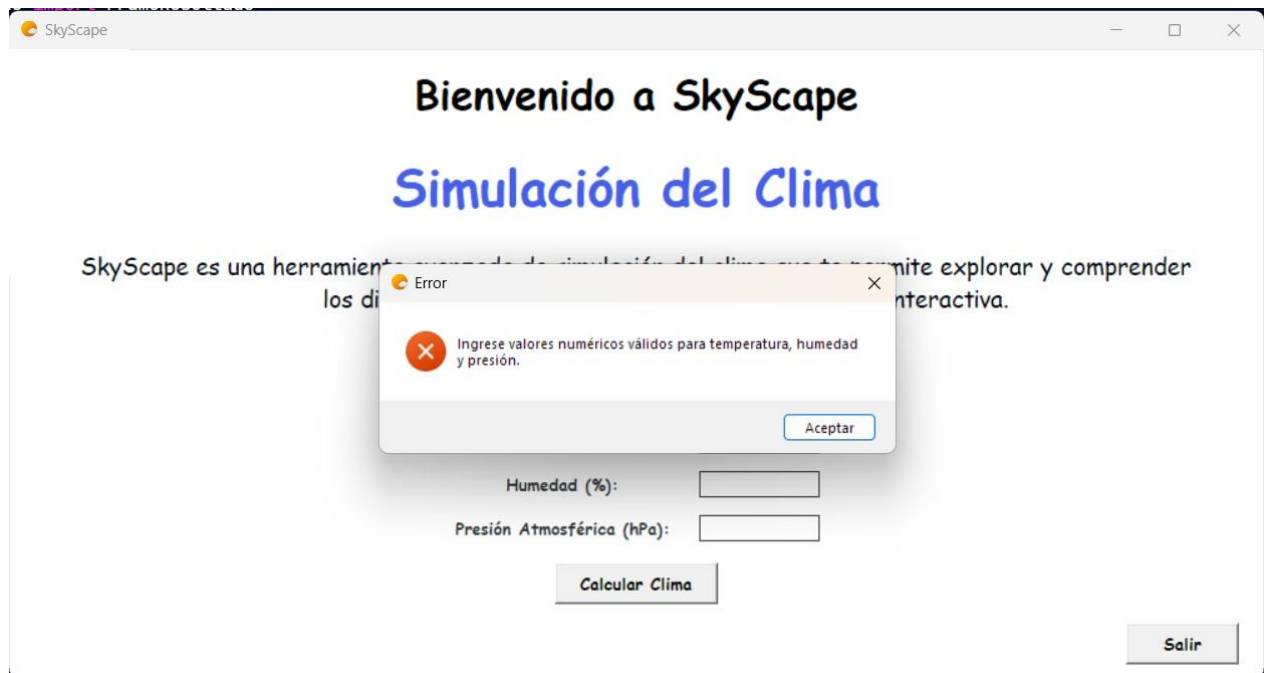
Fecha: 23/05/2024

Temperatura: 15°C

Humedad: 65%

Presión: 1012 hPa

5.4 Prueba 4. Si no se ingresan datos suficientes o datos en general.



5.5 Prueba 5. Otro clima

SkyScape

Bienvenido a SkyScape

Simulación del Clima

SkyScape es una herramienta avanzada de simulación del clima que te permite explorar y comprender los diferentes fenómenos meteorológicos de forma interactiva.

Ciudad:

FRANCIA

Fecha (DD/MM/AAAA):

01/08/2024

Temperatura (°C):

23

Humedad (%):

75


Presión Atmosférica (hPa):

1000

Calcular Clima

Salir

SkyScape - Simulación del clima



El clima es: Lluvioso

Ciudad: FRANCIA

Fecha: 01/08/2024

Temperatura: 23°C

Humedad: 75%

Presión: 1000 hPa

6. DECISIÓN DE DISEÑO O CONSIDERACION IMPORTANTE

En este caso, la decisión de diseño fue abarcar los temas vistos en clase para el desarrollo de nuestro proyecto final, con el objetivo de facilitar la utilidad del código y favorecer la extensibilidad, permitiendo adaptarse a cambios futuros del sistema.

7. PILARES DE POO IMPLEMENTADOS

Nuestro objetivo principal era implementar los cuatro pilares fundamentales de la programación orientada a objetos. Este enfoque nos ha permitido desarrollar un simulador climático robusto y adaptable, capaz de escalar y evolucionar con nuevas funcionalidades y mejoras de rendimiento. Esta estrategia se alinea perfectamente con los principios de diseño de software, fomentando la extensibilidad y reutilización del código, y fortaleciendo la cohesión y el bajo acoplamiento en nuestro diseño final.

8. CONCLUSIONES

El desarrollo de nuestro simulador de clima ha sido una experiencia enriquecedora que ha fomentado habilidades creativas y la resolución de problemas. Este proyecto nos ha brindado la oportunidad invaluable de aplicar conocimientos teóricos en un contexto práctico y significativo, poniendo a prueba los temas aprendidos en clase. Estamos orgullosos del resultado alcanzado y confiamos en que este trabajo no solo fortalecerá nuestro entendimiento del clima, sino también nuestra capacidad para enfrentar desafíos con ingenio y determinación.