Universidad: Universidad Autónoma de Entre Ríos.

Facultad: Facultad de Ciencia y Tecnología.

Carrera: Licenciatura en Sistemas de Información.

Asignatura: Ingeniería de Software II

Equipo Docente: Dr. Pedro E. Colla.

Alumnos Integrantes: Rubén Zeni.

Fecha límite de entrega: 07/05/2025.

Trabajo Práctico N°4: Patrones Estructurales.

1. Provea una clase ping que luego de creada al ser invocada con un método “execute(string)” realice 10 intentos de ping a la dirección IP contenida en “string” (argumento pasado), la clase solo debe funcionar si la dirección IP provista comienza con “192.”. Provea un método executefree(string) que haga lo mismo pero sin el control de dirección. Ahora provea una clase pingproxy cuyo método execute(string) si la dirección es “192.168.0.254” realice un ping a www.google.com usando el método executefree de ping y reenvíe a execute de la clase ping en cualquier otro caso. (Modele la solución como un patrón proxy).
2. Para un producto láminas de acero de 0.5” de espesor y 1,5 metros de ancho dispone de dos trenes laminadores, uno que genera planchas de 5 mts y otro de 10 mts. Genere una clase que represente a las láminas en forma genérica al cual se le pueda indicar que a que tren laminador se enviará a producir. (Use el patrón bridge en la solución).
3. Represente la lista de piezas componentes de un ensamblado con sus relaciones jerárquicas. Empiece con un producto principal formado por tres subconjuntos los que a su vez tendrán cuatro piezas cada uno. Genere clases que representen esa configuración y la muestren. Luego agregue un subconjunto opcional adicional también formado por cuatro piezas. (Use el patrón composite).
4. Implemente una clase que permita a un número cualquiera imprimir su valor, luego agregarle sucesivamente.
   a. Sumarle 2.
   b. Multiplicarle por 2.
   c. Dividirlo por 3.

   Mostrar los resultados de la clase sin agregados y con la invocación anidada a las clases con las diferentes operaciones. Use un patrón Decorator para implementar.

5. Imagine una situación donde pueda ser de utilidad el patrón “flyweight”.

1.

Resumen

- **Ping:** sólo hace ping cuando la IP comienza con “192.” (método **execute**), o sin control (método **executefree**).
- **PingProxy:** si la IP es **192.168.0.254** redirige a **www.google.com** usando **executefree** de **Ping**; en otro caso, delega a **execute** de **Ping**.

```
import subprocess

class Ping:
    def __init__(self, attempts: int = 10):
        self.attempts = attempts

    def execute(self, ip: str):
        """Sólo hace ping si la IP comienza con '192.'"""
        if not ip.startswith("192."):
            raise ValueError(f"Dirección no permitida: {ip}")
        return self._do_ping(ip)

    def executefree(self, ip: str):
        """Hace ping sin restricción de IP."""
        return self._do_ping(ip)

    def _do_ping(self, ip: str):
        cmd = ["ping", "-c", str(self.attempts), ip]
        print(f"Haciendo ping a {ip} ({self.attempts} intentos)...")
        return subprocess.run(cmd, capture_output=True, text=True).stdout

class PingProxy:
    def __init__(self):
        self._ping = Ping()

    def execute(self, ip: str):
            """Si la IP es 192.168.0.254, redirige a www.google.com vía ejecutefree; si no,
delega a execute."""
        if ip == "192.168.0.254":
            print("Proxy interceptó IP crítica; usando google.com sin restricción.")
            return self._ping.executefree("www.google.com")
        else:
            print("Proxy delega ping normal.")
            return self._ping.execute(ip)

# Ejemplo de uso:
if __name__ == "__main__":
```

```
proxy = PingProxy()
try:
    salida1 = proxy.execute("192.168.0.254")
    print(salida1)
    salida2 = proxy.execute("192.168.1.10")
    print(salida2)
    # Este último lanzará excepción en Ping.execute
    proxy.execute("10.0.0.5")
except ValueError as e:
    print(f"Error: {e}")
```

2.

Resumen

- Abstracción: Laminate representa la lámina genérica.
- Implementador: RollingMill es la interfaz de tren laminador; dos implementaciones (Mill5m, Mill10m).
- La lámina delega la producción al tren elegido.

```
from abc import ABC, abstractmethod

# Implementador
class RollingMill(ABC):
    @abstractmethod
    def produce(self, thickness: float, width: float) -> str:
        pass

class Mill5m(RollingMill):
    def produce(self, thickness: float, width: float) -> str:
        return (f"Producida lámina de {thickness}\" x {width}m "
                f"en tren de 5m")

class Mill10m(RollingMill):
    def produce(self, thickness: float, width: float) -> str:
        return (f"Producida lámina de {thickness}\" x {width}m "
                f"en tren de 10m")

# Abstracción
class Laminate:
    def __init__(self, thickness: float, width: float, mill: RollingMill):
        self.thickness = thickness
        self.width = width
        self.mill = mill

    def set_mill(self, mill: RollingMill):
```

```
        self.mill = mill

    def produce(self) -> str:
        return self.mill.produce(self.thickness, self.width)

# Ejemplo de uso:
if __name__ == "__main__":
    lam = Laminate(0.5, 1.5, Mill5m())
    print(lam.produce())   # Tren 5m
    lam.set_mill(Mill10m())
    print(lam.produce())   # Ahora tren 10m
```

3.

Resumen

- Component: interfaz común para piezas y conjuntos.
- Leaf: pieza indivisible.
- Composite: subconjunto que agrupa componentes.

```
from abc import ABC, abstractmethod

class Component(ABC):
    @abstractmethod
    def show(self, indent: int = 0):
        pass

class Leaf(Component):
    def __init__(self, name: str):
        self.name = name

    def show(self, indent: int = 0):
        print(" " * indent + f"- Pieza: {self.name}")

class Composite(Component):
    def __init__(self, name: str):
        self.name = name
        self._children = []

    def add(self, component: Component):
        self._children.append(component)

    def remove(self, component: Component):
        self._children.remove(component)
```

```
    def show(self, indent: int = 0):
        print(" " * indent + f"+ Conjunto: {self.name}")
        for child in self._children:
            child.show(indent + 4)

# Generación de la jerarquía:
if __name__ == "__main__":
    producto = Composite("Producto Principal")
    # Tres subconjuntos iniciales
    for i in range(1, 4):
        sub = Composite(f"Subconjunto {i}")
        # Cada uno con cuatro piezas
        for j in range(1, 5):
            sub.add(Leaf(f"Pieza {i}.{j}"))
        producto.add(sub)
    # Mostrar antes de opcional
    print("Estructura inicial:")
    producto.show()

    # Agregar subconjunto opcional adicional
    opcional = Composite("Subconjunto Opcional")
    for k in range(1, 5):
        opcional.add(Leaf(f"Pieza O.{k}"))
    producto.add(opcional)

    print("\nEstructura con subconjunto opcional:")
    producto.show()
```

4. Resumen
   - Base: Number imprime y almacena valor.
   - Decorators: Add2, Mul2, Div3 añaden operaciones sucesivas.

```
from abc import ABC, abstractmethod

# Componente
class Number(ABC):
    @abstractmethod
    def get(self) -> float:
        pass
```

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

# Componente concreto
class SimpleNumber(Number):
    def __init__(self, value: float):
        self._value = value

    def get(self) -> float:
        return self._value

    def print(self):
        print(f"Valor: {self._value}")

# Decorator base
class NumberDecorator(Number):
    def __init__(self, wrapped: Number):
        self._wrapped = wrapped

    @abstractmethod
    def get(self) -> float:
        pass

    def print(self):
        print(f"Valor decorado: {self.get()}")

# Decorators concretos
class Add2(NumberDecorator):
    def get(self) -> float:
        return self._wrapped.get() + 2

class Mul2(NumberDecorator):
    def get(self) -> float:
        return self._wrapped.get() * 2

class Div3(NumberDecorator):
    def get(self) -> float:
        return self._wrapped.get() / 3

# Ejemplo de uso:
if __name__ == "__main__":
    base = SimpleNumber(10)
    print("Sin decoradores:")
    base.print()
```

```
# Encadenado: ((10 + 2) * 2) / 3
decorated = Div3(Mul2(Add2(base)))
print("\nCon decoradores anidados (Add2 → Mul2 → Div3):")
decorated.print()
```

5.

- **Escenario:** En un editor de texto que presenta miles de caracteres en pantalla, usaríamos Flyweight para compartir la representación interna (forma, tipografía, estilos básicos) de cada glifo y así reducir la memoria. Cada carácter dibujado referenciaría un objeto GlyphFlyweight compartido según tipo de letra y carácter, mientras que su posición y color serían datos extrínsecos.
- **Contexto:** Visualización de texto con millones de caracteres.
- **Flyweight:** objeto compartido por cada (carácter, fuente, estilo).
- **Estado intrínseco:** forma vectorial del glifo, métricas de fuente.
- **Estado extrínseco:** posición en pantalla, color, opacidad.