Softrware Patterns

1. Adaptive Pattern:

Definition:

The Adapter pattern is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two objects. Converts the interface of a class into another interface the client expects. Used when an existing class's interface does not match the interface expected by the client.

Implementation:

- Define the target interface that the client expects.
- Create an adapter class to implement the target interface, and internally use an instance of the incompatible class.
- The adapter translates client calls into calls to the adaptee.

Motivation:

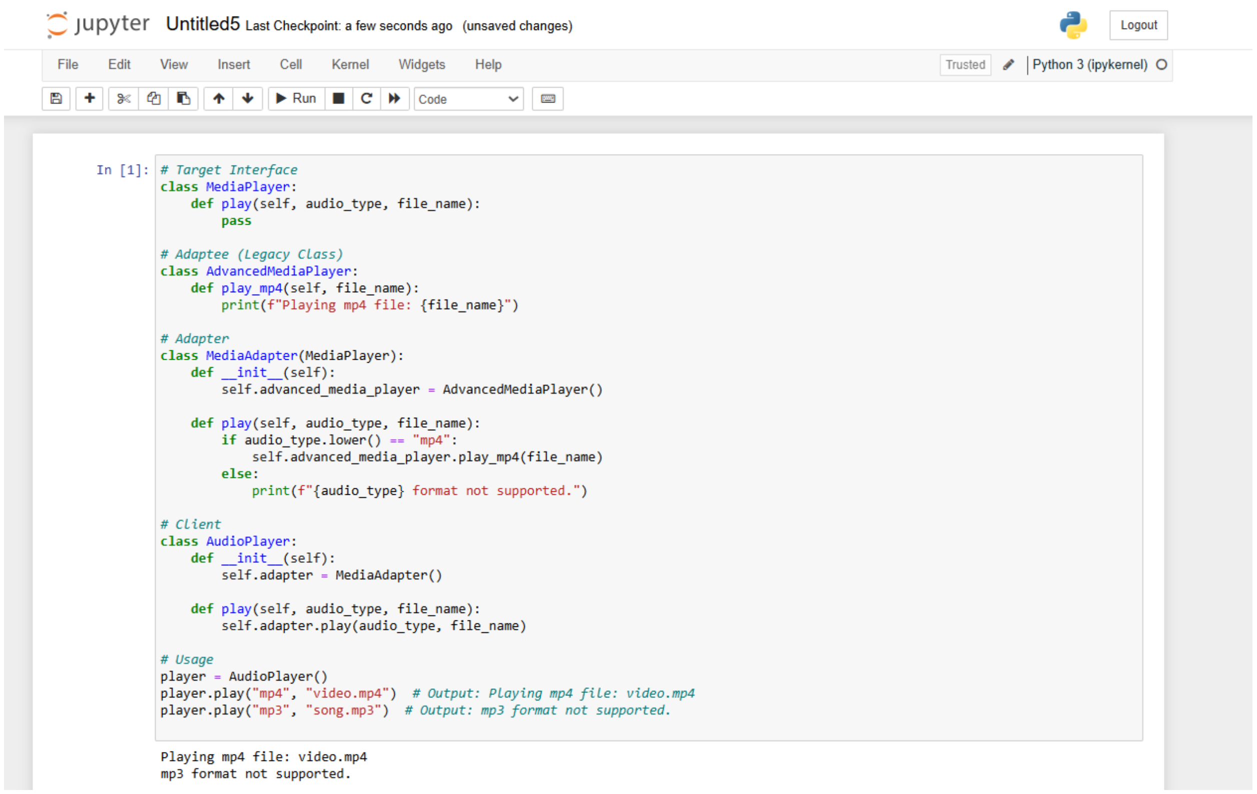
- Often, in software development, we encounter scenarios where an existing class's interface does not match the interface expected by the system or client.
- The Adapter pattern allows objects with incompatible interfaces to collaborate without changing their source code.

Examples:

- **Media Player:**

```
# Target Interface
class MediaPlayer:
    def play(self, audio_type, file_name):
        pass
# Adaptee (Legacy Class)
class AdvancedMediaPlayer:
    def play_mp4(self, file_name):
        print(f"Playing mp4 file: {file_name}")
# Adapter
class MediaAdapter(MediaPlayer):
    def __init__(self):
        self.advanced_media_player = AdvancedMediaPlayer()
    def play(self, audio_type, file_name):
        if audio_type.lower() == "mp4":
            self.advanced_media_player.play_mp4(file_name)
        else:
            print(f"{audio_type} format not supported.")
class AudioPlayer:
    def __init__(self):
        self.adapter = MediaAdapter()
    def play(self, audio_type, file_name):
        self.adapter.play(audio_type, file_name)
player = AudioPlayer()
player.play("mp4", "video.mp4")  # Output: Playing mp4 file: video.mp4
player.play("mp3", "song.mp3")  # Output: mp3 format not supported.
```

Output:


jupyter Untitled5 Last Checkpoint: a few seconds ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code

In [1]:
```
# Target Interface
class MediaPlayer:
    def play(self, audio_type, file_name):
        pass

# Adaptee (Legacy Class)
class AdvancedMediaPlayer:
    def play_mp4(self, file_name):
        print(f"Playing mp4 file: {file_name}")

# Adapter
class MediaAdapter(MediaPlayer):
    def __init__(self):
        self.advanced_media_player = AdvancedMediaPlayer()

    def play(self, audio_type, file_name):
        if audio_type.lower() == "mp4":
            self.advanced_media_player.play_mp4(file_name)
        else:
            print(f"{audio_type} format not supported.")

# Client
class AudioPlayer:
    def __init__(self):
        self.adapter = MediaAdapter()

    def play(self, audio_type, file_name):
        self.adapter.play(audio_type, file_name)

# Usage
player = AudioPlayer()
player.play("mp4", "video.mp4")  # Output: Playing mp4 file: video.mp4
player.play("mp3", "song.mp3")  # Output: mp3 format not supported.
```

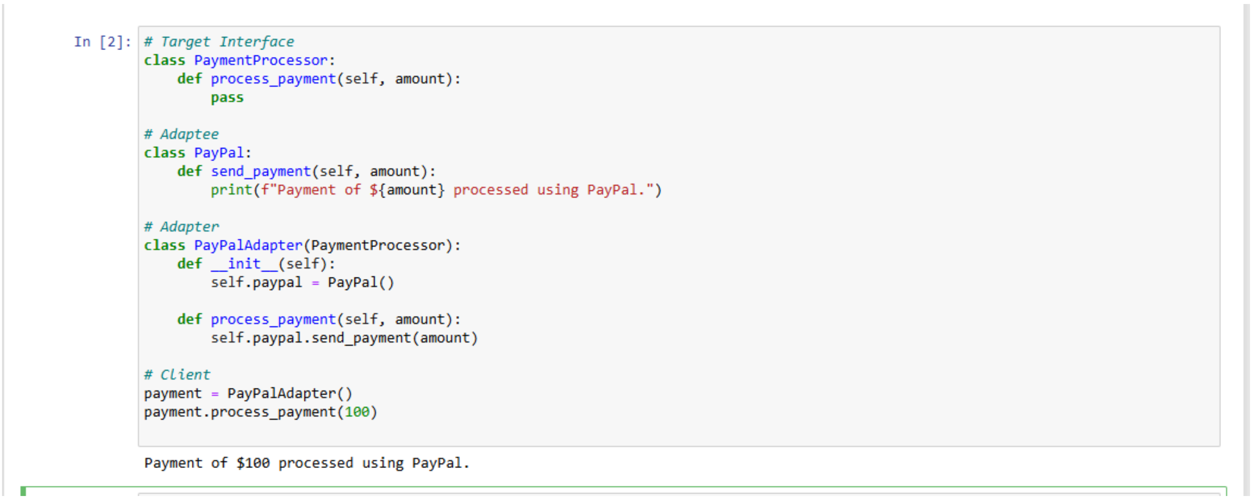
```
Playing mp4 file: video.mp4
mp3 format not supported.
```

- **Adapter for Payment Systems:**

```
# Target Interface

class PaymentProcessor:

    def process_payment(self, amount):

        pass

# Adaptee

class PayPal:

    def send_payment(self, amount):

        print(f"Payment of ${amount} processed using PayPal.")

# Adapter
```

```
class PayPalAdapter(PaymentProcessor):
   def __init__(self):
        self.paypal = PayPal()
    def process_payment(self, amount):
        self.paypal.send_payment(amount)
# Client
payment = PayPalAdapter()
payment.process_payment(100)
```

Output:



```
In [2]: # Target Interface
        class PaymentProcessor:
            def process_payment(self, amount):
                pass

        # Adaptee
        class PayPal:
            def send_payment(self, amount):
                print(f"Payment of ${amount} processed using PayPal.")

        # Adapter
        class PayPalAdapter(PaymentProcessor):
            def __init__(self):
                self.paypal = PayPal()

            def process_payment(self, amount):
                self.paypal.send_payment(amount)

        # Client
        payment = PayPalAdapter()
        payment.process_payment(100)

        Payment of $100 processed using PayPal.
```

2. Prototype pattern:

Definition:

The Prototype pattern is a creational design pattern that allows cloning of objects, even complex ones, without coupling to their specific classes Specifies the kinds of objects to create using a prototypical instance and create new objects by copying the prototype. Used when object creation is costly, and a prototype instance is available.

Implementation:

- Define a prototype interface with a cloning method.
- Implement this interface in the classes that need to be cloned.
- Use the cloning method to create copies.

Motivation:

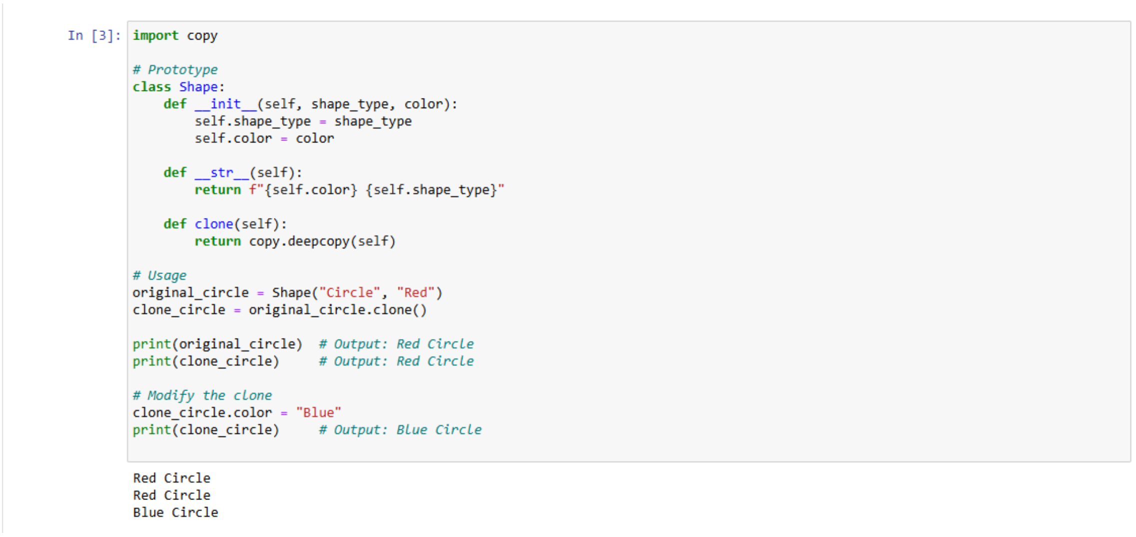
- Object creation can sometimes be expensive due to complex initialization, requiring numerous resources or database queries. Instead of creating new objects from scratch, the Prototype pattern allows you to clone existing objects.
- This is especially useful when the specific type of object to be created is determined at runtime.

Examples:

- **Shape Cloning:**

```
import copy
# Prototype
class Shape:
    def __init__(self, shape_type, color):
        self.shape_type = shape_type
        self.color = color
    def __str__(self):
        return f"{self.color} {self.shape_type}"
    def clone(self):
        return copy.deepcopy(self)
# Usage
original_circle = Shape("Circle", "Red")
clone_circle = original_circle.clone()
print(original_circle)  # Output: Red Circle
print(clone_circle)     # Output: Red Circle
# Modify the clone
clone_circle.color = "Blue"
print(clone_circle)     # Output: Blue Circle
```

Output:

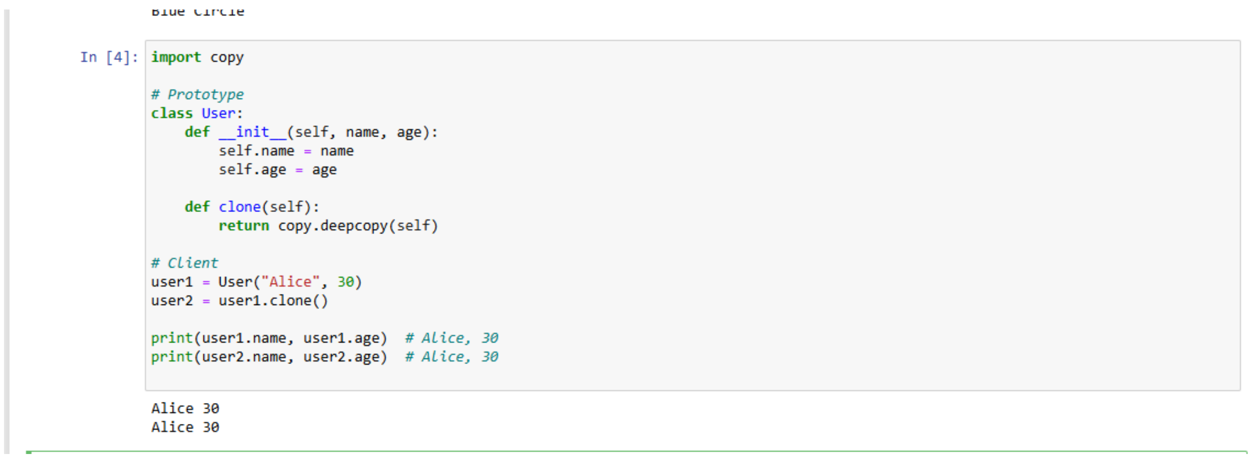


```
In [3]: import copy

        # Prototype
        class Shape:
            def __init__(self, shape_type, color):
                self.shape_type = shape_type
                self.color = color

            def __str__(self):
                return f"{self.color} {self.shape_type}"

            def clone(self):
                return copy.deepcopy(self)

        # Usage
        original_circle = Shape("Circle", "Red")
        clone_circle = original_circle.clone()

        print(original_circle)  # Output: Red Circle
        print(clone_circle)     # Output: Red Circle

        # Modify the clone
        clone_circle.color = "Blue"
        print(clone_circle)     # Output: Blue Circle

        Red Circle
        Red Circle
        Blue Circle
```

- **Cloning a User Object:**

```
import copy
# Prototype
class User:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def clone(self):
    return copy.deepcopy(self)
# Client
user1 = User("Alice", 30)
user2 = user1.clone()
print(user1.name, user1.age)  # Alice, 30
print(user2.name, user2.age)  # Alice, 30
```

Output:



```
Blue Circle

In [4]: import copy

        # Prototype
        class User:
            def __init__(self, name, age):
                self.name = name
                self.age = age

            def clone(self):
                return copy.deepcopy(self)

        # Client
        user1 = User("Alice", 30)
        user2 = user1.clone()

        print(user1.name, user1.age)  # Alice, 30
        print(user2.name, user2.age)  # Alice, 30

        Alice 30
        Alice 30
```

3. Decorator Pattern

Definition:

The Decorator Pattern is a structural design pattern that allows you to dynamically add new behaviors or responsibilities to objects at runtime by wrapping them in "decorator" classes. Decorators provide a flexible alternative to subclassing for extending functionality.

Implementation:

- Defines the interface for objects that can have additional behaviors.
- The base implementation of the component.
- Implements the component interface and contains a reference to a component object to which it delegates requests.
- Extend the functionalities of the component by adding new behaviors.

Motivation:

Sometimes you want to add functionality to objects in a flexible way without using inheritance. For instance:

- You want to add scrolling to a text box.
- You want to add encryption to a file object.

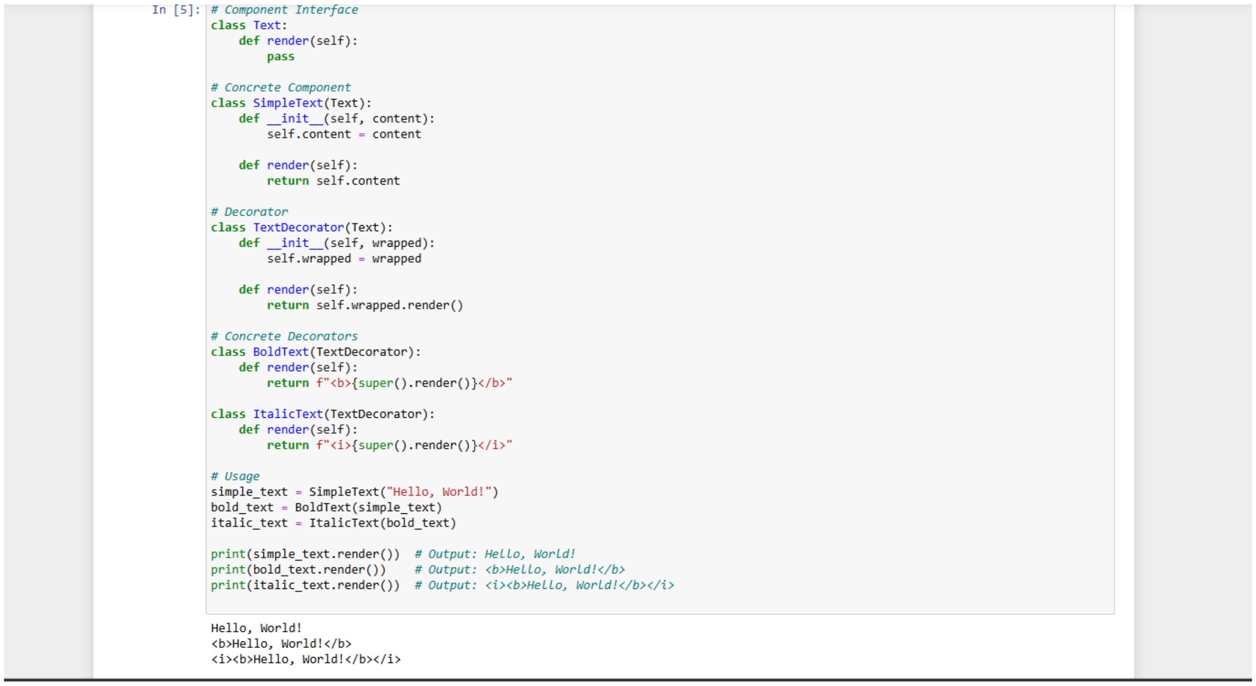
Instead of modifying existing classes or creating numerous subclasses to support every possible combination of behaviors, you can use decorators to wrap objects dynamically.

Examples:

- **Text Formatter:**

```
# Component Interface
class Text:
    def render(self):
        pass
# Concrete Component
class SimpleText(Text):
    def __init__(self, content):
        self.content = content
    def render(self):
        return self.content
# Decorator
class TextDecorator(Text):
    def __init__(self, wrapped):
        self.wrapped = wrapped
    def render(self):
        return self.wrapped.render()
# Concrete Decorators
class BoldText(TextDecorator):
    def render(self):
        return f"<b>{super().render()}</b>"
class ItalicText(TextDecorator):
    def render(self):
        return f"<i>{super().render()}</i>"
# Usage
simple_text = SimpleText("Hello, World!")
bold_text = BoldText(simple_text)
italic_text = ItalicText(bold_text)
print(simple_text.render())  # Output: Hello, World!
print(bold_text.render())    # Output: <b>Hello, World!</b>
print(italic_text.render())  # Output: <i><b>Hello, World!</b></i>
```

Output:

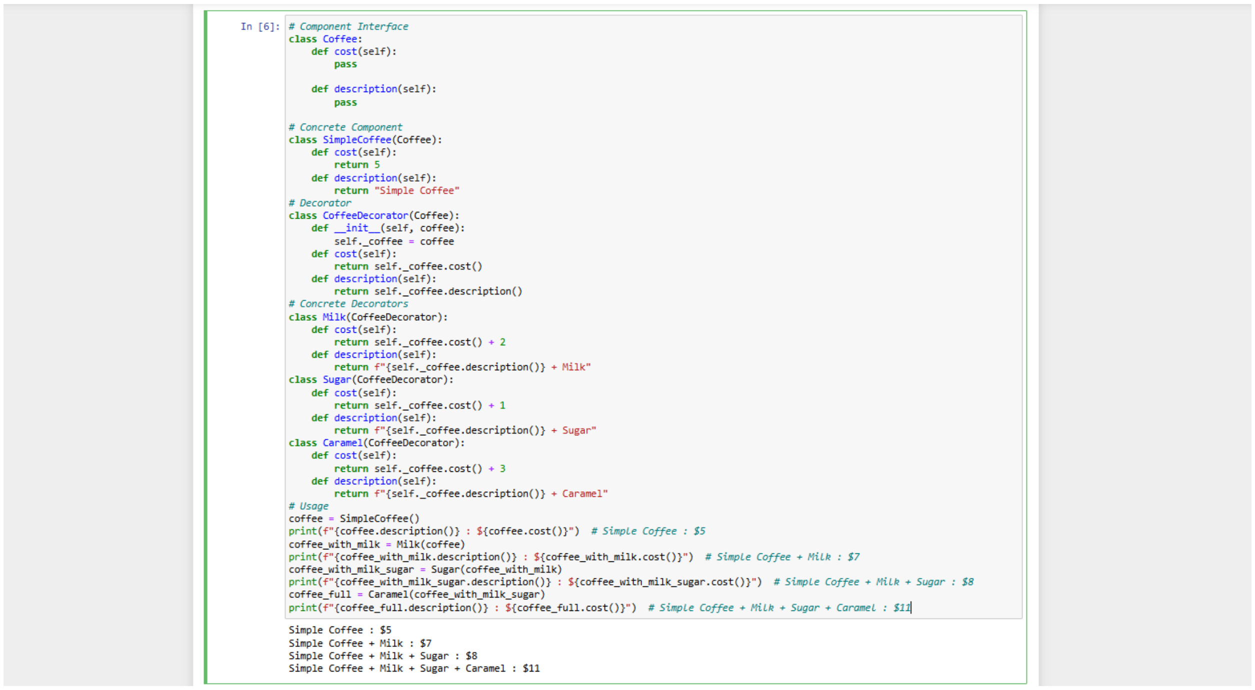


```
In [5]: # Component Interface
        class Text:
            def render(self):
                pass

        # Concrete Component
        class SimpleText(Text):
            def __init__(self, content):
                self.content = content

            def render(self):
                return self.content

        # Decorator
        class TextDecorator(Text):
            def __init__(self, wrapped):
                self.wrapped = wrapped

            def render(self):
                return self.wrapped.render()

        # Concrete Decorators
        class BoldText(TextDecorator):
            def render(self):
                return f"<b>{super().render()}</b>"

        class ItalicText(TextDecorator):
            def render(self):
                return f"<i>{super().render()}</i>"

        # Usage
        simple_text = SimpleText("Hello, World!")
        bold_text = BoldText(simple_text)
        italic_text = ItalicText(bold_text)

        print(simple_text.render())  # Output: Hello, World!
        print(bold_text.render())    # Output: <b>Hello, World!</b>
        print(italic_text.render())  # Output: <i><b>Hello, World!</b></i>

        Hello, World!
        <b>Hello, World!</b>
        <i><b>Hello, World!</b></i>
```

- **Coffee Shop:**

```
# Component Interface
class Coffee:
    def cost(self):
        pass
    def description(self):
        pass
# Concrete Component
class SimpleCoffee(Coffee):
    def cost(self):
        return 5
    def description(self):
        return "Simple Coffee"
# Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee
    def cost(self):
        return self._coffee.cost()
    def description(self):
        return self._coffee.description()
# Concrete Decorators
class Milk(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 2
    def description(self):
        return f"{self._coffee.description()} + Milk"
class Sugar(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 1
    def description(self):
        return f"{self._coffee.description()} + Sugar"
class Caramel(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 3
    def description(self):
        return f"{self._coffee.description()} + Caramel"
# Usage
coffee = SimpleCoffee()
print(f"{coffee.description()} : ${coffee.cost()}")  # Simple Coffee : $5
coffee_with_milk = Milk(coffee)
print(f"{coffee_with_milk.description()} : ${coffee_with_milk.cost()}")  # Simple Coffee + Milk : $7
coffee_with_milk_sugar = Sugar(coffee_with_milk)
print(f"{coffee_with_milk_sugar.description()} : ${coffee_with_milk_sugar.cost()}")  # Simple Coffee
+ Milk + Sugar : $8
coffee_full = Caramel(coffee_with_milk_sugar)
print(f"{coffee_full.description()} : ${coffee_full.cost()}")  # Simple Coffee + Milk + Sugar + Caramel :
$11
```

Output:



```
In [6]: # Component Interface
class Coffee:
    def cost(self):
        pass

    def description(self):
        pass

# Concrete Component
class SimpleCoffee(Coffee):
    def cost(self):
        return 5
    def description(self):
        return "Simple Coffee"
# Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee
    def cost(self):
        return self._coffee.cost()
    def description(self):
        return self._coffee.description()
# Concrete Decorators
class Milk(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 2
    def description(self):
        return f"{self._coffee.description()} + Milk"
class Sugar(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 1
    def description(self):
        return f"{self._coffee.description()} + Sugar"
class Caramel(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 3
    def description(self):
        return f"{self._coffee.description()} + Caramel"
# Usage
coffee = SimpleCoffee()
print(f"{coffee.description()} : ${coffee.cost()}")  # Simple Coffee : $5
coffee_with_milk = Milk(coffee)
print(f"{coffee_with_milk.description()} : ${coffee_with_milk.cost()}")  # Simple Coffee + Milk : $7
coffee_with_milk_sugar = Sugar(coffee_with_milk)
print(f"{coffee_with_milk_sugar.description()} : ${coffee_with_milk_sugar.cost()}")  # Simple Coffee + Milk + Sugar : $8
coffee_full = Caramel(coffee_with_milk_sugar)
print(f"{coffee_full.description()} : ${coffee_full.cost()}")  # Simple Coffee + Milk + Sugar + Caramel : $11

Simple Coffee : $5
Simple Coffee + Milk : $7
Simple Coffee + Milk + Sugar : $8
Simple Coffee + Milk + Sugar + Caramel : $11
```