DESIGN PATTERNS - CHAPITRE 10

Decorator en Python

Ajouter des responsabilités à un objet en préservant son interface par délégation.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Ajouter des responsabilités à un objet en préservant son interface par délégation.
- Prévoir l'ordre d'enveloppement et conserver les propriétés attendues.

Prérequis. Savoir utiliser classes, héritage, composition, listes, dictionnaires et exceptions Python. Les mécanismes spécifiques sont expliqués avant leur emploi.

Decorator : intention et expérience

Decorator : ajouter un comportement par enveloppement

Le patron **Decorator** enveloppe un objet en conservant son interface. Lait et sucre délèguent au café puis ajoutent leur description et leur coût. Les enveloppes sont combinables sans créer une sous-classe pour chaque combinaison. Comparez deux ordres pour observer ce qui change.

Le patron objet Decorator et la syntaxe Python `@decorator` sont distincts : on peut réaliser le premier sans utiliser la seconde.

Python

```python
# example.py
# Component
class Coffee:
    def get_cost(self):
        return 5

    def get_description(self):
        return "Simple coffee"

# Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self.coffee = coffee

    def get_cost(self):
        return self.coffee.get_cost()

    def get_description(self):
        return self.coffee.get_description()

# Concrete Decorator
class MilkDecorator(CoffeeDecorator):
    def get_cost(self):
        return super().get_cost() + 2

    def get_description(self):
        return super().get_description() + ", with milk"

# Concrete Decorator
class SugarDecorator(CoffeeDecorator):
    def get_cost(self):
        return super().get_cost() + 1

    def get_description(self):
        return super().get_description() + ", with sugar"

# Client code
```

```
if __name__ == "__main__":
    coffee = Coffee()
    print(f"{coffee.get_description()} Cost: ${coffee.get_cost()}")

    milk_coffee = MilkDecorator(coffee)
    print(f"{milk_coffee.get_description()} Cost: ${milk_coffee.get_cost()}")

    sugar_milk_coffee = SugarDecorator(milk_coffee)
    print(f"{sugar_milk_coffee.get_description()} Cost:
${sugar_milk_coffee.get_cost()}")
```

Décorer une méthode avec des fonctions

Les deux fonctions suivantes reçoivent une fonction et renvoient une enveloppe. La première transforme son résultat en majuscules, la seconde échappe les caractères HTML puis ajoute un paragraphe. `escape` vient du module standard `html`.

Python

```python
# example.py
from functools import wraps

from html import escape

class TextFormatter:
    """
    Base class for text formatting.
    Initializes with the text to be formatted.
    """
    def __init__(self, text):
        self._text = text

    def format(self):
        """
        Returns the formatted text.
        """
        return self._text


def uppercase_decorator(func):
    """
    Decorator to convert text to uppercase.
    It wraps the original function and modifies its output.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        """
        Wrapper function that converts the result to uppercase.
        """
        formatted_text = func(*args, **kwargs)
        return formatted_text.upper()
    return wrapper


def html_decorator(func):
    """
```

```
    Decorator to wrap text in HTML paragraph tags.
    It adds paragraph tags around escaped text.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        """
        Wrapper function that adds HTML paragraph tags.
        """
        formatted_text = func(*args, **kwargs)
        return f"<p>{escape(formatted_text)}</p>"
    return wrapper
```

Suite du bloc précédent : définissez les deux décorateurs avant cette classe. Les décorateurs sont appliqués de bas en haut : la méthode est d'abord enveloppée par la transformation en majuscules, puis par l'ajout du paragraphe. Essayez un texte contenant `<` pour observer l'échappement.

Python

```python
# example.py
class TextFormatter:
    """
    Base class for text formatting.
    Initializes with the text to be formatted.
    """
    def __init__(self, text):
        self._text = text

    @html_decorator
    @uppercase_decorator
    def format(self):
        """
        Returns the formatted text.
        Now decorated with uppercase and HTML formatting.
        """
        return self._text

# Example usage
text_formatter = TextFormatter("Hello, world!")
formatted_text = text_formatter.format()
print(formatted_text)  # Output: <p>HELLO, WORLD!</p>
```

Vérifier et interpréter

Inversez lait et sucre : le coût reste identique mais l'ordre de la description change. Un enrichissement peut donc être commutatif pour une propriété et pas pour une autre. Chaque enveloppe doit déléguer le contrat complet ; multiplier les couches rend le chemin d'un appel plus difficile à suivre.

Version web : https://www.clearcode.fr/courses/programming/design-patterns/structure/decorator