DESIGN PATTERNS - CHAPITRE 30

Comparer les intentions et les contrats

Choisir selon l'intention plutôt que selon la ressemblance du code.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Distinguer identité, construction et collaboration.
- Comparer des interfaces similaires avec des comportements différents.

Prérequis. Avoir étudié les patrons de création, de structure et de comportement.

Comparer les intentions et les contrats

Singleton ou fabrique : deux questions différentes

Singleton contrôle la réutilisation d'une identité ; une fabrique simple choisit une classe à construire. Ces intentions peuvent coexister, mais l'une ne résout pas le problème de l'autre. Le `Factory` ci-dessous effectue une sélection conditionnelle : ce n'est pas le Factory Method présenté plus haut.

Python

```python
# example.py
# Example illustrating the difference between Singleton and Simple Factory

# Singleton Pattern
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

class Configuration:
    def __init__(self, setting1, setting2):
        self.setting1 = setting1
        self.setting2 = setting2

# Example usage of Singleton
config1 = Singleton()
config1.configuration = Configuration("value1", "value2")

config2 = Singleton()
print(config2.configuration.setting1)  # Accessing the same configuration
instance

# Simple Factory Pattern
class AbstractProduct:
    def operation(self):
        raise NotImplementedError("Subclasses must implement this method")

class ConcreteProductA(AbstractProduct):
    def operation(self):
        return "Product A"

class ConcreteProductB(AbstractProduct):
    def operation(self):
        return "Product B"
```

```python
class Factory:
    def create_product(self, product_type):
        if product_type == "A":
            return ConcreteProductA()
        elif product_type == "B":
            return ConcreteProductB()
        else:
            raise ValueError("Unknown product type")

# Example usage of Simple Factory
factory = Factory()
product_a = factory.create_product("A")
print(product_a.operation())
```

Observer ou Strategy : plusieurs réactions ou une politique

Observer distribue une notification à plusieurs abonnés. Strategy délègue un calcul à une politique choisie. Les deux stratégies suivantes renvoient une liste, mais réalisent volontairement des opérations différentes : tri croissant et inversion de l'ordre initial. Une interface commune ne signifie pas des résultats identiques.

Python

```python
# example.py
# Observer Pattern Example

class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self, event):
        for observer in self._observers:
            observer.update(event)

class Observer:
    def update(self, event):
        raise NotImplementedError("Subclasses must implement this method")

class ConcreteObserverA(Observer):
    def update(self, event):
        print(f"Observer A received event: {event}")

class ConcreteObserverB(Observer):
    def update(self, event):
        print(f"Observer B handled event: {event}")

# Strategy Pattern Example

class Strategy:
    def execute(self, data):
        raise NotImplementedError("Subclasses must implement this method")

class ConcreteStrategyA(Strategy):
    def execute(self, data):
```

```
        return sorted(data)  # Example: Sort data

class ConcreteStrategyB(Strategy):
    def execute(self, data):
        return list(reversed(data)) # Example: Reverse data

class Context:
    def __init__(self, strategy):
        self._strategy = strategy

    def set_strategy(self, strategy):
        self._strategy = strategy

    def process_data(self, data):
        return self._strategy.execute(data)

# Example Usage:
subject = Subject()
observer_a = ConcreteObserverA()
observer_b = ConcreteObserverB()

subject.attach(observer_a)
subject.attach(observer_b)

subject.notify("New event occurred")

data = [5, 2, 8, 1, 9]
context = Context(ConcreteStrategyA())
sorted_data = context.process_data(data)
print(f"Sorted Data: {sorted_data}")

context.set_strategy(ConcreteStrategyB())
reversed_data = context.process_data(data)
print(f"Reversed Data: {list(reversed_data)}")
```

Adapter ou Decorator : compatibilité ou enrichissement

Adapter traduit une interface existante vers celle attendue. Decorator conserve l'interface pour ajouter un comportement. Identifiez le contrat attendu par le client avant de choisir. Les appels ci-dessous illustrent cette différence, même si les deux solutions contiennent une délégation.

Python

```python
# example.py
# Adapter Pattern Example

class Adaptee:
    def specific_request(self):
        return ".eetpadA"

class Target:
    def request(self):
        return "epdatdA"

class Adapter(Target):
    def __init__(self, adaptee):
        self._adaptee = adaptee

    def request(self):
        return self._adaptee.specific_request()[::-1]

# Decorator Pattern Example

class Component:
    def operation(self):
        return "Component"

class Decorator(Component):
    def __init__(self, component):
        self._component = component

    def operation(self):
        return self._component.operation()

class ConcreteDecoratorA(Decorator):
    def operation(self):
        return f"ConcreteDecoratorA({super().operation()})"

class ConcreteDecoratorB(Decorator):
    def operation(self):
```

```
        return f"ConcreteDecoratorB({super().operation()})"

# Example Usage
adaptee = Adaptee()
adapter = Adapter(adaptee)
print(adapter.request())

component = Component()
decorator_a = ConcreteDecoratorA(component)
decorator_b = ConcreteDecoratorB(decorator_a)

print(decorator_b.operation())
```

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