DESIGN PATTERNS - CHAPITRE 29

S'exercer puis comparer sa solution

Concevoir puis vérifier des promotions, des notifications et des équipements composables.

Par Ali ZAINOUL · Cours gratuit

Ce que vous allez apprendre

- Tester les limites des réductions et les désabonnements.
- Conserver toutes les propriétés attendues dans une chaîne de décorateurs.

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

S'exercer puis comparer sa solution

Exercice 1 : appliquer une promotion

Écrivez un contrat `apply_discount(total)` qui renvoie la réduction, pas le prix final. Testez une réduction fixe supérieure au total, une réduction de 0 %, puis de 100 %. Refusez les montants négatifs et les pourcentages hors de cet intervalle. Les nombres ci-dessous sont des unités de simulation ; aucun arrondi monétaire n'est modélisé.

▼ Voir une correction

Python

```python
# example.py
class Order:
    def __init__(self, total, promotion=None):
        if total < 0:
            raise ValueError("Total must be non-negative")
        self.total = total
        self.promotion = promotion

    def calculate_discount(self):
        if self.promotion:
            return self.promotion.apply_discount(self.total)
        return 0

    def calculate_final_price(self):
        discount = self.calculate_discount()
        return self.total - discount

class Promotion:
    def apply_discount(self, total):
        raise NotImplementedError

class FixedDiscount(Promotion):
    def __init__(self, discount_amount):
        if discount_amount < 0:
            raise ValueError("Discount must be non-negative")
        self.discount_amount = discount_amount

    def apply_discount(self, total):
        return min(self.discount_amount, total)

class PercentageDiscount(Promotion):
    def __init__(self, percentage):
        if not 0 <= percentage <= 100:
            raise ValueError("Percentage must be between 0 and 100")
        self.percentage = percentage / 100

    def apply_discount(self, total):
```

```
        return total * self.percentage

# Example Usage
order1 = Order(100, FixedDiscount(10))
print(f"Final price for order 1: {order1.calculate_final_price()}") #
Output: Final price for order 1: 90

order2 = Order(100, PercentageDiscount(20))
print(f"Final price for order 2: {order2.calculate_final_price()}") #
Output: Final price for order 2: 80.0

order3 = Order(100)
print(f"Final price for order 3: {order3.calculate_final_price()}") #
Output: Final price for order 3: 100
```

Exercice 2 : notifier un changement de prix

Un prix inchangé ne doit pas produire de notification. Avec deux abonnés, une modification doit prévenir chacun une fois ; après désabonnement, seul l'autre reçoit les modifications. Proposez vos tests avant de consulter la correction. Les prix sont fictifs.

▼ Voir une correction

Python

```python
# example.py
class Stock:
    def __init__(self, symbol, price):
        self.symbol = symbol
        self.price = price
        self._observers = []

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

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

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

    def set_price(self, new_price):
        if self.price != new_price:
            self.price = new_price
            self.notify()

class Investor:
    def __init__(self, name):
        self.name = name

    def update(self, stock):
        print(f"{self.name} received update: {stock.symbol} price changed
to {stock.price}")

# Example Usage
stock = Stock("AAPL", 150)
investor1 = Investor("Alice")
investor2 = Investor("Bob")

stock.attach(investor1)
stock.attach(investor2)
```

```
stock.set_price(155) # Output: Alice received update: AAPL price changed to
155
                     # Output: Bob received update: AAPL price changed to
155

stock.detach(investor1)
stock.set_price(160) # Output: Bob received update: AAPL price changed to
160
```

Exercice 3 : composer des équipements

Ajoutez une armure puis un pouvoir magique au même personnage. Toutes les enveloppes doivent conserver les opérations et propriétés attendues, notamment `health` et `attack`. Une armure doit donc déléguer l'attaque, et un pouvoir doit déléguer la santé. Ici, les descriptions indiquent les bonus ; les propriétés donnent les statistiques calculées.

▼ **Voir une correction**

Python

```python
# example.py
class GameCharacter:
    def __init__(self, name, health, attack):
        self.name = name
        self.health = health
        self.attack = attack

    def get_description(self):
        return f"{self.name} - Health: {self.health}, Attack:
{self.attack}"

class CharacterDecorator:
    def __init__(self, character):
        self.character = character

    @property
    def name(self):
        return self.character.name

    @property
    def health(self):
        return self.character.health

    @property
    def attack(self):
        return self.character.attack

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

class ArmorDecorator(CharacterDecorator):
    def __init__(self, character, armor_points):
        super().__init__(character)
        self.armor_points = armor_points

    def get_description(self):
        return f"{super().get_description()}, Armor: +{self.armor_points}"
```

```
    @property
    def health(self):
        return self.character.health + self.armor_points

class MagicPowerDecorator(CharacterDecorator):
    def __init__(self, character, magic_attack):
        super().__init__(character)
        self.magic_attack = magic_attack

    def get_description(self):
        return f"{super().get_description()}, Magic Attack: +
{self.magic_attack}"

    @property
    def attack(self):
        return self.character.attack + self.magic_attack

# Example Usage
hero = GameCharacter("Hero", 100, 20)
print(hero.get_description()) # Output: Hero - Health: 100, Attack: 20

armored_hero = ArmorDecorator(hero, 30)
print(armored_hero.get_description()) # Output: Hero - Health: 100, Attack:
20, Armor: +30
print(f"Armored Hero health: {armored_hero.health}") # Output: Armored Hero
health: 130

magic_hero = MagicPowerDecorator(hero, 15)
print(magic_hero.get_description()) # Output: Hero - Health: 100, Attack:
20, Magic Attack: +15
print(f"Magic Hero attack: {magic_hero.attack}") # Output: Magic Hero
attack: 35
```

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