**Observer:**

from abc import ABC, abstractmethod

from typing import List

# Interface cho Observer (người dùng đăng ký nhận thông báo)

class NotificationObserver(ABC):

@abstractmethod

def update(self, message: str):

pass

# Interface cho Subject (dịch vụ thông báo)

class NotificationService(ABC):

@abstractmethod

def register\_observer(self, observer: NotificationObserver):

pass

@abstractmethod

def remove\_observer(self, observer: NotificationObserver):

pass

@abstractmethod

def notify\_observers(self, message: str):

pass

# Triển khai cụ thể của NotificationService

class ConcreteNotificationService(NotificationService):

def \_\_init\_\_(self):

self.\_observers: List[NotificationObserver] = []

def register\_observer(self, observer: NotificationObserver):

if observer not in self.\_observers:

self.\_observers.append(observer)

print(f"Đã đăng ký người dùng: {observer.name}")

def remove\_observer(self, observer: NotificationObserver):

if observer in self.\_observers:

self.\_observers.remove(observer)

print(f"Đã hủy đăng ký người dùng: {observer.name}")

def notify\_observers(self, message: str):

print(f"\nGửi thông báo mới: {message}")

for observer in self.\_observers:

observer.update(message)

# Triển khai cụ thể của Observer (người dùng)

class User(NotificationObserver):

def \_\_init\_\_(self, name: str):

self.name = name

def update(self, message: str):

print(f"Người dùng {self.name} nhận được thông báo: {message}")

# Client code

if \_\_name\_\_ == "\_\_main\_\_":

# Tạo dịch vụ thông báo

notification\_service = ConcreteNotificationService()

# Tạo các người dùng

user1 = User("Alice")

user2 = User("Bob")

user3 = User("Charlie")

# Đăng ký người dùng nhận thông báo

notification\_service.register\_observer(user1)

notification\_service.register\_observer(user2)

# Gửi thông báo đầu tiên

notification\_service.notify\_observers("Có bản cập nhật mới phiên bản 2.0!")

# Đăng ký thêm người dùng mới

notification\_service.register\_observer(user3)

# Gửi thông báo thứ hai

notification\_service.notify\_observers("Khuyến mãi đặc biệt trong 24h!")

# Hủy đăng ký một người dùng

notification\_service.remove\_observer(user1)

# Gửi thông báo thứ ba

notification\_service.notify\_observers("Tin nhắn mới từ quản trị viên")

**Strategy:**

from abc import ABC, abstractmethod

from typing import Protocol

# Interface cho các chiến lược tính thuế

class TaxStrategy(Protocol):

def calculate\_tax(self, amount: float) -> float:

pass

# Các chiến lược cụ thể

class VATStrategy:

def calculate\_tax(self, amount: float) -> float:

return amount \* 0.1 # Thuế VAT 10%

class ImportTaxStrategy:

def calculate\_tax(self, amount: float) -> float:

return amount \* 0.15 # Thuế nhập khẩu 15%

class LuxuryTaxStrategy:

def calculate\_tax(self, amount: float) -> float:

return amount \* 0.2 # Thuế hàng xa xỉ 20%

# Lớp sản phẩm sử dụng Strategy

class Product:

def \_\_init\_\_(self, name: str, price: float, tax\_strategy: TaxStrategy):

self.name = name

self.base\_price = price

self.tax\_strategy = tax\_strategy

def calculate\_total\_price(self) -> float:

tax = self.tax\_strategy.calculate\_tax(self.base\_price)

return self.base\_price + tax

def set\_tax\_strategy(self, tax\_strategy: TaxStrategy):

self.tax\_strategy = tax\_strategy

def \_\_str\_\_(self):

total = self.calculate\_total\_price()

return f"{self.name} - Giá gốc: {self.base\_price:.2f} - Tổng: {total:.2f}"

# Client code

if \_\_name\_\_ == "\_\_main\_\_":

# Tạo các chiến lược tính thuế

vat = VATStrategy()

import\_tax = ImportTaxStrategy()

luxury\_tax = LuxuryTaxStrategy()

# Tạo sản phẩm với các chiến lược thuế khác nhau

laptop = Product("Laptop ABC", 1000, vat)

iphone = Product("iPhone XYZ", 1500, import\_tax)

watch = Product("Đồng hồ Rolex", 5000, luxury\_tax)

print("--- Giá sản phẩm ban đầu ---")

print(laptop)

print(iphone)

print(watch)

# Thay đổi chiến lược thuế tại runtime

print("\n--- Sau khi thay đổi thuế ---")

laptop.set\_tax\_strategy(import\_tax)

iphone.set\_tax\_strategy(luxury\_tax)

print(laptop)

print(iphone)

print(watch)

# Thêm chiến lược thuế mới không cần sửa code hiện có

class SpecialTaxStrategy:

def calculate\_tax(self, amount: float) -> float:

return amount \* 0.05 # Thuế đặc biệt 5%

print("\n--- Áp dụng thuế đặc biệt mới ---")

special\_tax = SpecialTaxStrategy()

laptop.set\_tax\_strategy(special\_tax)

print(laptop)

**Decorator:**

from abc import ABC, abstractmethod

# Interface cơ bản cho đơn hàng

class Order(ABC):

@abstractmethod

def get\_description(self) -> str:

pass

@abstractmethod

def get\_cost(self) -> float:

pass

# Lớp cơ bản đại diện cho đơn hàng

class BasicOrder(Order):

def \_\_init\_\_(self, description: str, cost: float):

self.\_description = description

self.\_cost = cost

def get\_description(self) -> str:

return self.\_description

def get\_cost(self) -> float:

return self.\_cost

# Lớp decorator cơ sở

class OrderDecorator(Order):

def \_\_init\_\_(self, decorated\_order: Order):

self.decorated\_order = decorated\_order

def get\_description(self) -> str:

return self.decorated\_order.get\_description()

def get\_cost(self) -> float:

return self.decorated\_order.get\_cost()

# Các decorator cụ thể

class InsuranceDecorator(OrderDecorator):

def \_\_init\_\_(self, decorated\_order: Order):

super().\_\_init\_\_(decorated\_order)

self.\_insurance\_cost = 50

def get\_description(self) -> str:

return f"{self.decorated\_order.get\_description()} + Bảo hiểm"

def get\_cost(self) -> float:

return self.decorated\_order.get\_cost() + self.\_insurance\_cost

class WarrantyDecorator(OrderDecorator):

def \_\_init\_\_(self, decorated\_order: Order):

super().\_\_init\_\_(decorated\_order)

self.\_warranty\_cost = 100

def get\_description(self) -> str:

return f"{self.decorated\_order.get\_description()} + Bảo hành 1 năm"

def get\_cost(self) -> float:

return self.decorated\_order.get\_cost() + self.\_warranty\_cost

class ExpressShippingDecorator(OrderDecorator):

def \_\_init\_\_(self, decorated\_order: Order):

super().\_\_init\_\_(decorated\_order)

self.\_shipping\_cost = 30

def get\_description(self) -> str:

return f"{self.decorated\_order.get\_description()} + Vận chuyển nhanh"

def get\_cost(self) -> float:

return self.decorated\_order.get\_cost() + self.\_shipping\_cost

class GiftWrapDecorator(OrderDecorator):

def \_\_init\_\_(self, decorated\_order: Order):

super().\_\_init\_\_(decorated\_order)

self.\_gift\_cost = 15

def get\_description(self) -> str:

return f"{self.decorated\_order.get\_description()} + Gói quà"

def get\_cost(self) -> float:

return self.decorated\_order.get\_cost() + self.\_gift\_cost

# Client code

if \_\_name\_\_ == "\_\_main\_\_":

# Tạo đơn hàng cơ bản

laptop\_order = BasicOrder("Laptop Dell XPS 15", 1500)

print(f"Đơn hàng cơ bản: {laptop\_order.get\_description()}")

print(f"Giá tiền: ${laptop\_order.get\_cost():.2f}\n")

# Thêm các tính năng bổ sung

laptop\_order = InsuranceDecorator(laptop\_order)

laptop\_order = WarrantyDecorator(laptop\_order)

print(f"Đơn hàng sau khi thêm bảo hiểm và bảo hành:")

print(f"Mô tả: {laptop\_order.get\_description()}")

print(f"Giá tiền: ${laptop\_order.get\_cost():.2f}\n")

# Thêm tiếp vận chuyển nhanh

laptop\_order = ExpressShippingDecorator(laptop\_order)

print(f"Đơn hàng sau khi thêm vận chuyển nhanh:")

print(f"Mô tả: {laptop\_order.get\_description()}")

print(f"Giá tiền: ${laptop\_order.get\_cost():.2f}\n")

# Tạo đơn hàng khác với các tính năng khác

phone\_order = BasicOrder("iPhone 15 Pro", 1200)

phone\_order = GiftWrapDecorator(phone\_order)

phone\_order = ExpressShippingDecorator(phone\_order)

print(f"Đơn hàng điện thoại với gói quà và vận chuyển nhanh:")

print(f"Mô tả: {phone\_order.get\_description()}")

print(f"Giá tiền: ${phone\_order.get\_cost():.2f}\n")

# Có thể kết hợp nhiều decorator theo nhiều cách

# Ví dụ thêm bảo hiểm sau khi đã có gói quà

phone\_order = InsuranceDecorator(phone\_order)

print(f"Đơn hàng sau khi thêm bảo hiểm:")

print(f"Mô tả: {phone\_order.get\_description()}")

print(f"Giá tiền: ${phone\_order.get\_cost():.2f}")

**Factory:**

from abc import ABC, abstractmethod

from enum import Enum

# Enum định nghĩa các loại sản phẩm

class ProductType(Enum):

PHONE = 1

LAPTOP = 2

TABLET = 3

# Interface chung cho tất cả sản phẩm

class Product(ABC):

@abstractmethod

def get\_info(self) -> str:

pass

@abstractmethod

def calculate\_price(self) -> float:

pass

# Các lớp sản phẩm cụ thể

class Phone(Product):

def \_\_init\_\_(self, name: str, base\_price: float):

self.name = name

self.base\_price = base\_price

self.tax\_rate = 0.1 # Thuế 10%

def get\_info(self) -> str:

return f"Điện thoại {self.name}"

def calculate\_price(self) -> float:

return self.base\_price \* (1 + self.tax\_rate)

class Laptop(Product):

def \_\_init\_\_(self, name: str, base\_price: float):

self.name = name

self.base\_price = base\_price

self.tax\_rate = 0.15 # Thuế 15%

self.warranty\_cost = 50 # Phí bảo hành

def get\_info(self) -> str:

return f"Laptop {self.name}"

def calculate\_price(self) -> float:

return self.base\_price \* (1 + self.tax\_rate) + self.warranty\_cost

class Tablet(Product):

def \_\_init\_\_(self, name: str, base\_price: float):

self.name = name

self.base\_price = base\_price

self.tax\_rate = 0.08 # Thuế 8%

self.accessory\_cost = 30 # Phụ kiện đi kèm

def get\_info(self) -> str:

return f"Tablet {self.name}"

def calculate\_price(self) -> float:

return self.base\_price \* (1 + self.tax\_rate) + self.accessory\_cost

# Factory class để tạo sản phẩm

class ProductFactory:

def create\_product(self, product\_type: ProductType, name: str, price: float) -> Product:

if product\_type == ProductType.PHONE:

return Phone(name, price)

elif product\_type == ProductType.LAPTOP:

return Laptop(name, price)

elif product\_type == ProductType.TABLET:

return Tablet(name, price)

else:

raise ValueError("Loại sản phẩm không hợp lệ")

# Client code

if \_\_name\_\_ == "\_\_main\_\_":

factory = ProductFactory()

# Tạo các sản phẩm khác nhau

products = [

factory.create\_product(ProductType.PHONE, "iPhone 15", 999),

factory.create\_product(ProductType.LAPTOP, "Dell XPS 15", 1599),

factory.create\_product(ProductType.TABLET, "iPad Pro", 799)

]

# Hiển thị thông tin sản phẩm

print("Danh sách sản phẩm:")

for product in products:

print(f"- {product.get\_info()}")

print(f" Giá bán: ${product.calculate\_price():.2f}")

# Thêm sản phẩm mới không cần sửa client code

class SmartWatch(Product):

def \_\_init\_\_(self, name: str, base\_price: float):

self.name = name

self.base\_price = base\_price

self.tax\_rate = 0.05

def get\_info(self) -> str:

return f"Smart Watch {self.name}"

def calculate\_price(self) -> float:

return self.base\_price \* (1 + self.tax\_rate)

# Mở rộng factory để hỗ trợ sản phẩm mới

class ExtendedProductFactory(ProductFactory):

def create\_product(self, product\_type: ProductType, name: str, price: float) -> Product:

if product\_type == ProductType(4): # Giả sử thêm ProductType.SMARTWATCH = 4

return SmartWatch(name, price)

else:

return super().create\_product(product\_type, name, price)

print("\nThêm sản phẩm mới:")

extended\_factory = ExtendedProductFactory()

watch = extended\_factory.create\_product(ProductType(4), "Apple Watch", 399)

print(f"- {watch.get\_info()}")

print(f" Giá bán: ${watch.calculate\_price():.2f}")