**ĐẠI HỌC QUỐC GIA TP.HCM**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**A blue and white logo

Description automatically generated with medium confidence**

ADVANCED PROGRAMMING PROJECT

**ĐỀ TÀI**

**DESIGN PATTERNS IN OOP**

**\_\_\_\_\_\_\*\*\*\*\*\*\*\*\*\*\_\_\_\_\_**

CLASS: **CN01** – **HK222**

GV hướng dẫn: **Trương Tuấn Anh**

Sinh viên thực hiện: **Bùi Hồ Hải Đăng - 2153289**

*Thành phố Hồ Chí Minh – 2022*

Table of Contents

[**CHƯƠNG 1** 3](#_Toc137672171)

[1.1. Design Patterns là gì? 3](#_Toc137672172)

[1.2. Có bao nhiêu loại patterns có trong OOP? 4](#_Toc137672173)

[**CHƯƠNG 2** 8](#_Toc137672174)

[2.1. Nhóm khởi tạo – Creational Design Pattern 9](#_Toc137672175)

[**Factory Method Design Pattern** 10](#_Toc137672176)

[**Abstract Factory Design Pattern** 16](#_Toc137672177)

[**Builder design pattern** 24](#_Toc137672178)

[**Prototype** 32](#_Toc137672179)

[**Singleton** 36](#_Toc137672180)

[2.1. Nhóm cấu trúc (Structural Patterns) 40](#_Toc137672181)

[**Adapter** 41](#_Toc137672182)

[**Bridge** 47](#_Toc137672183)

[**Composite** 53](#_Toc137672184)

[**Decorator** 59](#_Toc137672185)

[**Facade** 65](#_Toc137672186)

[**Flyweight** 71](#_Toc137672187)

[**Proxy** 77](#_Toc137672188)

[2.3. Nhóm Behavioral (nhóm hành vi/ tương tác) 83](#_Toc137672189)

[**Chain of Responsibility** 84](#_Toc137672190)

[**Command** 90](#_Toc137672191)

[**Interpreter** 97](#_Toc137672192)

[**Iterator** 103](#_Toc137672193)

[**Mediator** 108](#_Toc137672194)

[**Memento** 114](#_Toc137672195)

[**Observer** 119](#_Toc137672196)

[**State** 125](#_Toc137672197)

[**Strategy** 130](#_Toc137672198)

[**Template Method** 136](#_Toc137672199)

[**Visitor** 142](#_Toc137672200)

[**REFERENCE** 149](#_Toc137672201)

# **CHƯƠNG 1**

**INTRODUCTION TO DESIGN PATTERNS IN OOP**

Việc thiết kế một phần mềm hướng đối tượng đã là khó, thì việc thiết kế một phần mềm hướng đối tượng có thể sử dụng lại thì thậm chí càng khó hơn. Khi nói đến phát triển phần mềm, chi phí và thời gian là những thước đo quan trọng. Thời gian phát triển ngắn hơn cho phép thâm nhập thị trường sớm hơn so với các đối thủ cạnh tranh, đồng thời giảm chi phí phát triển để lại nhiều tiền hơn cho hoạt động tiếp thị và mở rộng phạm vi tiếp cận với khách hàng tiềm năng. Và khi đó thuật ngữ **design patterns** xuất hiện.

## 1.1. Design Patterns là gì?

**Định nghĩa**

Design Pattern là một giải pháp tổng quát và có thể tái sử dụng cho một vấn đề phổ biến trong một ngữ cảnh cụ thể trong thiết kế phần mềm. Đó không phải là một thiết kế hoàn chỉnh có thể được chuyển đổi trực tiếp thành mã nguồn hoặc máy. Thay vào đó, đó là một mô tả hoặc mẫu cho cách giải quyết một vấn đề có thể được sử dụng trong nhiều tình huống khác nhau.

“Mỗi Pattern mô tả một vấn đề xảy ra lặp đi lặp lại trong môi trường của chúng ta, và sau đó mô tả cốt lõi của giải pháp cho vấn đề đó, theo cách mà bạn có thể sử dụng giải pháp này hàng triệu lần mà không bao giờ thực hiện theo cùng một cách hai lần.” – theo Christopher Alexander.

Bằng cách sử dụng các design patterns, các nhà phát triển có thể hưởng lợi từ kinh nghiệm tập thể hoặc bản thân của các lập trình viên khác, những người đã đối mặt với những thách thức tương tự. Các mẫu thiết kế cung cấp một ngôn ngữ chung và sự hiểu biết chung về cách giải quyết vấn đề, điều này có thể giúp các nhà phát triển làm việc cùng nhau hiệu quả dễ dàng hơn.

<https://sourcemaking.com/design_patterns>

**Tác dụng của Design Patterns**

1. Khuyến khích việc tái sử dụng mã: Design pattern (mẫu thiết kế) khuyến khích sử dụng mã có thể tái sử dụng bằng cách cung cấp các giải pháp đã được chứng minh cho các vấn đề lập trình thường gặp. Điều này có thể tiết kiệm thời gian và công sức khi phát triển ứng dụng phần mềm mới.
2. Cải thiện tính dễ đọc và tổ chức của code: Design pattern (mẫu thiết kế) cung cấp một ngôn ngữ chung cho các nhà phát triển để trao đổi về thiết kế phần mềm cũng như các kiến trúc sư có thể quen thuộc với các pattern. Điều này có thể giúp làm cho mã dễ đọc, dễ hiểu và dễ sửa đổi.
3. Giúp phát hiện các vấn đề tiềm ẩn sớm: Bằng cách tuân thủ các mẫu thiết kế đã được thiết lập, các nhà phát triển có thể tránh các lỗ hổng và vấn đề tiềm ẩn trong mã của họ. Điều này có thể giúp ngăn chặn lỗi và các vấn đề lớn khác xuất hiện trong quá trình phát triển sau này.
4. Thuận tiện cho việc hợp tác giữa các nhà phát triển: Design pattern (mẫu thiết kế) cung cấp một từ vựng chung cho các nhà phát triển để trao đổi về thiết kế phần mềm, điều này có thể giúp các thành viên trong nhóm làm việc với nhau một cách hiệu quả.
5. Khuyến khích tính mở rộng và bảo trì phần mềm: Bằng cách sử dụng các mẫu thiết kế, các nhà phát triển có thể tạo ra phần mềm có tính mô-đun và dễ cập nhật và bảo trì trong thời gian. Điều này có thể giúp đảm bảo rằng phần mềm vẫn hoạt động và phù hợp trong nhiều năm tới cũng như khiến chúng trở nên mạnh mẽ hơn so với các thiết kế đặc biệt.

## 1.2. Có bao nhiêu loại patterns có trong OOP?

Trong lập trình hướng đối tượng (OOP), có tổng cộng là 23 mẫu thiết kế được công nhận rộng rãi và được chia làm 3 mục lớn: *Creational Design Patterns, Structural Design Patterns, Behavioral Design Patterns.*

**Nhóm khởi tạo – Creational Design Pattern**

*Những loại design patterns này đều xoay quanh về khởi tạo các class trong OOP. Những patterns này được chia thành các class-creation patterns và các objects-creation patterns. Trong khi các class-creation patterns sử dụng inheritance một cách hiệu quả trong quá trình khởi tạo thì object-creation patterns sử dụng các delegation một cách hiệu quả để hoàn thành công việc.*

* Abstract Factory

Tạo ra một object (đối tượng) của một số họ của các class.

* Builder

Tách cấu trúc của object khỏi biểu diễn của nó.

* Factory Method

Tạo một object của một số class dẫn xuất.

* Prototype

Một object được khởi tạo đầy đủ để được sao chép hoặc nhân bản.

* Singleton

Một class mà nó chỉ tồn tại một object duy nhất.

**Nhóm cấu trúc – Structural Design Patterns**

*Những loại design patterns này đều liên quan đến việc sáng tạo class và object. Những Structural class-creation patterns có cấu trúc inheritance để sáng tạo các interface. Những structural object-creation patterns có cấu trúc define các cách để sáng tạo cho các object đạt được các chức năng mới.*

* Adapter

Kết hợp interface của các class khác nhau.

* Bridge

Tách interface của object khỏi việc triển khai nó.

* Composite

Một cấu trúc cây của các object đơn giản và hỗn hợp.

* Decorator

Thêm trách nhiệm cho các object một cách linh hoạt.

* Façade

Một class duy nhất đại diện cho toàn bộ subsystem.

* Flyweight

Một object chi tiết được sử dụng để chia sẻ hiệu quả.

* Proxy

Một object đại diện cho một object khác.

**Nhóm hành vi – Behavioral Design Patterns**

*Những loại design patterns này xoay quanh việc truyền thông giữa các object của class. Behavioral design patterns là những mẫu thiết kế đặc biệt quan tâm đến việc truyền thông giữa các objects.*

* Interpreter

Một cách để bao gồm các yếu tố ngôn ngữ trong một chương trình.

* Template method

Trì hoãn các bước chính xác của thuật toán cho một class con.

* Chain of Responsibility

Một cách để chuyển yêu cầu giữa một chuỗi các object.

* Command

Đóng gói một yêu cầu lệnh như một object.

* Iterator

Truy cập tuần tự các phần tử của một bộ sưu tập.

* Mediator

Định nghĩa giao tiếp đơn giản giữa các class.

* Memento

Chụp và khôi phục trạng thái bên trong của object.

* Observer

Một cách thông báo thay đổi một số lượng các class.

* State

Thay đổi hành vi của một object khi trạng thái của nó thay đổi.

* Strategy

Đóng gói một thuật toán bên trong một class.

* Visitor

Định nghĩa một hoạt động mới cho một class mà không thay đổi.

**Mối quan hệ giữa 23 Design Pattern cơ bản (GoF):**

A picture containing text, diagram, line, plan

Description automatically generated

# **CHƯƠNG 2**

**DESCRIPTION DESIGN PATTERNS IN OOP**

Năm 1994, bốn tác giả **Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides** đã cho xuất bản một cuốn sách với tiêu đề **Design Patterns – Elements of Reusable Object-Oriented Software**, đây là khởi nguồn của khái niệm design pattern trong lập trình phần mềm.

Với việc đã tìm hiểu và biết được sự phân loại cũng như sơ lược về các loại design patterns của OOP ở trên, ta sẽ tìm hiểu sâu hơn về từng loại design patterns.

## 2.1. Nhóm khởi tạo – Creational Design Pattern

(nhóm khởi tạo – 5 mẫu) gồm: *Factory Method, Abstract Factory, Builder, Prototype, Singleton*. Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method **new**. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.

### **Factory Method Design Pattern**

https://viblo.asia/p/factory-method-design-pattern-tro-thu-dac-luc-cua-developers-924lJBLYlPM

**Giới thiệu:**

* Factory Method là một mẫu thiết kế (design pattern) thuộc nhóm Creational Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Factory Method là cung cấp một giao diện chung để tạo đối tượng, nhưng cho phép các class con quyết định loại đối tượng cụ thể cần tạo.
* Khi sử dụng Factory Method, chúng ta không khởi tạo đối tượng trực tiếp bằng cách sử dụng toán tử "new" như trong cách tiếp cận thông thường. Thay vào đó, chúng ta định nghĩa một phương thức gọi là "factory method" trong một class cơ sở (base class) hoặc một giao diện chung (interface). Class con sẽ triển khai phương thức này để tạo ra đối tượng cụ thể tương ứng.
* Tần suất sử dụng: cao.

**Mục tiêu:** Xác định một giao diện để tạo một đối tượng, nhưng hãy để các class con quyết định class nào sẽ khởi tạo. Factory Method cho phép một class trì hoãn việc khởi tạo thành các class con.

**Áp dụng:** Ta sử dụng Factory Method Pattern khi:

* Một class không thể biết trước class đối tượng mà nó phải tạo.
* Một class muốn các class con của nó chỉ định các đối tượng mà nó tạo ra.
* Các class ủy thác trách nhiệm cho một trong số các class con của trình trợ giúp và bạn muốn bản địa hóa kiến thức về class con của trình trợ giúp nào được ủy quyền.

**Các thành phần chính:**

* Product (Document): định nghĩa giao diện của các đối tượng được tạo bởi phương thức của nhà máy.
* ConcreteProduct (MyDocument): triển khai giao diện Product.
* Creator (Application):

+ Khai báo phương thức nhà máy, trả về một đối tượng của kiểu Product. Creator cũng có thể xác định một cài đặt mặc định của phương thức nhà máy trả về một đối tượng ConcreteProduct mặc định.

+ Có thể gọi phương thức nhà máy để tạo một đối tượng Product.

* ConcreteCreator (MyApplication): ghi đè phương thức nhà máy để trả về một phiên bản của ConcreteProduct.

**Ưu điểm:**

* Che giấu quá trình xử lý logic của phương thức khởi tạo
* Hạn chế sự phụ thuộc giữa creator và concrete products
* Dễ dàng mở rộng, thêm những đoạn code mới vào chương trình mà không cần phá vỡ các đối tượng ban đầu
* Giúp gom các đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác.
* Giảm khả năng gây lỗi compile, trong trường hợp chúng ta cần tạo một đối tượng mà quên khai báo class, chúng ta cũng có thể xử lý lỗi trong Factory và khai báo class cho chúng sau.

=> Vì những đặc điểm trên nên factory pattern thường được sử dụng trong các thư viện (người sử dụng đạt được mục đích tạo mới object và không cần quan tâm đến cách nó được tạo ra)

**Nhược điểm:**

* Source code có thể trở nên phức tạp hơn mức bình thường do đòi hỏi phải sử dụng nhiều class mới có thể cài đặt được pattern này.
* Việc refactoring ( tái cấu trúc ) một class bình thường có sẵn thành một class có Factory Method có thể dẫn đến nhiều lỗi trong hệ thống, phá vỡ sự tồn tại của clients.
* Factory method pattern lệ thuộc vào việc sử dụng private constructor nên các class không thể mở rộng và kế thừa.

**Top of Form**

**Cấu trúc:**

A diagram of a process

Description automatically generated with low confidence

**Template đối với Python:**

"""

Define an interface for creating an object, but let subclasses decide

which class to instantiate. Factory Method lets a class defer

instantiation to subclasses.

"""

import abc

class Creator(metaclass=abc.ABCMeta):

    """

    Declare the factory method, which returns an object of type Product.

    Creator may also define a default implementation of the factory

    method that returns a default ConcreteProduct object.

    Call the factory method to create a Product object.

    """

    def \_\_init\_\_(self):

        self.product = self.\_factory\_method()

    @abc.abstractmethod

    def \_factory\_method(self):

        pass

    def some\_operation(self):

        self.product.interface()

class ConcreteCreator1(Creator):

    """

    Override the factory method to return an instance of a

    ConcreteProduct1.

    """

    def \_factory\_method(self):

        return ConcreteProduct1()

class ConcreteCreator2(Creator):

    """

    Override the factory method to return an instance of a

    ConcreteProduct2.

    """

    def \_factory\_method(self):

        return ConcreteProduct2()

class Product(metaclass=abc.ABCMeta):

    """

    Define the interface of objects the factory method creates.

    """

    @abc.abstractmethod

    def interface(self):

        pass

class ConcreteProduct1(Product):

    """

    Implement the Product interface.

    """

    def interface(self):

        pass

class ConcreteProduct2(Product):

    """

    Implement the Product interface.

    """

    def interface(self):

        pass

def main():

    concrete\_creator = ConcreteCreator1()

    concrete\_creator.product.interface()

    concrete\_creator.some\_operation()

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

    main()

**Ví dụ:**

import abc

class Animal(metaclass=abc.ABCMeta):

    """

    Define the interface for creating an animal object.

    """

    @abc.abstractmethod

    def speak(self):

        pass

class Dog(Animal):

    """

    Implement the Animal interface for a Dog object.

    """

    def speak(self):

        return "Woof!"

class Cat(Animal):

    """

    Implement the Animal interface for a Cat object.

    """

    def speak(self):

        return "Meow!"

class AnimalCreator(metaclass=abc.ABCMeta):

    """

    Declare the factory method, which returns an object of type Animal.

    AnimalCreator may also define a default implementation of the factory

    method that returns a default Animal object.

    Call the factory method to create an Animal object.

    """

    def \_\_init\_\_(self):

        self.animal = self.\_create\_animal()

    @abc.abstractmethod

    def \_create\_animal(self):

        pass

    def do\_speak(self):

        return self.animal.speak()

class DogCreator(AnimalCreator):

    """

    Override the factory method to return an instance of a Dog.

    """

    def \_create\_animal(self):

        return Dog()

class CatCreator(AnimalCreator):

    """

    Override the factory method to return an instance of a Cat.

    """

    def \_create\_animal(self):

        return Cat()

def main():

    dog\_creator = DogCreator()

    dog = dog\_creator.do\_speak()

    print(dog)  # Output: Woof!

    cat\_creator = CatCreator()

    cat = cat\_creator.do\_speak()

    print(cat)  # Output: Meow!

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

    main()

**Giải thích code:**

* Trước tiên, chúng ta có giao diện Animal (Động vật) với phương thức speak (nói). Đây là một giao diện trừu tượng, vì vậy chúng ta chỉ định phương thức này mà không đưa ra triển khai cụ thể.
* Sau đó, chúng ta có hai class cụ thể là Dog (Chó) và Cat (Mèo). Cả hai class này đều triển khai giao diện Animal và cung cấp triển khai cho phương thức speak. Ví dụ, class Dog trả về chuỗi "Woof!" và class Cat trả về chuỗi "Meow!".
* Tiếp theo, chúng ta có class trừu tượng AnimalCreator (Người tạo động vật). Class này khai báo phương thức factory method \_create\_animal (tạo động vật) là một phương thức trừu tượng, có nhiệm vụ tạo ra một đối tượng động vật cụ thể. Ngoài ra, class còn có phương thức do\_speak (thực hiện nói) để gọi phương thức speak của đối tượng động vật được tạo ra.
* Cuối cùng, chúng ta có hai class cụ thể DogCreator và CatCreator, cả hai class này đều là class con của AnimalCreator. Các class con này ghi đè phương thức factory method \_create\_animal để trả về các đối tượng chó (Dog) và mèo (Cat) tương ứng.
* Trong hàm main(), chúng ta tạo một đối tượng DogCreator và gọi phương thức do\_speak() để lấy âm thanh của chó. Tương tự, chúng ta tạo một đối tượng CatCreator và lấy âm thanh của mèo. Kết quả đầu ra sẽ là "Woof!" và "Meow!" tương ứng.

### **Abstract Factory Design Pattern**

https://viblo.asia/p/abstract-factory-design-pattern-tro-thu-dac-luc-cua-developers-maGK7B4M5j2

**Giới thiệu:**

* Abstract Factory là một mẫu thiết kế (design pattern) thuộc nhóm Creational Patterns trong lĩnh vực phát triển phần mềm. Nó cung cấp một giao diện (interface) cho việc tạo ra các đối tượng có liên quan hoặc phụ thuộc nhau mà không cần chỉ định cụ thể các class cụ thể của chúng.
* Mục tiêu chính của Abstract Factory là tạo ra các đối tượng liên quan nhau mà không cần biết cụ thể các class cụ thể đang được sử dụng. Thay vì tạo ra các đối tượng trực tiếp bằng cách sử dụng từ khóa "new", Abstract Factory tạo ra các đối tượng thông qua một giao diện chung, cho phép các class cụ thể con thực hiện việc tạo ra các đối tượng cụ thể tương ứng.
* Abstract Factory còn được gọi là Kit
* Tuần suất sử dụng cao.

**Mục tiêu:** Cung cấp một giao diện (interface) để tạo các họ đối tượng liên quan hoặc đối tượng phụ thuộc mà không chỉ định các class cụ thể của chúng.

**Áp dụng:** ta sử dụng Abstract Factory Pattern khi

* Hệ thống nên độc lập với cách các sản phẩm của nó được tạo ra, cấu thành và thể hiện.
* Hệ thống nên được cấu hình với một trong những họ của sản phẩm.
* Một họ các đối tượng sản phẩm liên quan với nhau được thiết kế để sử dụng cùng nhau, và bạn thực thi các ràng buộc này.
* Bạn muốn cung cấp một thư viên class của các sản phẩm, và bạn chỉ muốn tiết lộ các giao diện chứ không phải các triển khai của chúng.

**Các thành phần chính:**

* AbstractFactory (WidgetFactory): khai báo một giao diện cho các hoạt động tạo đối tượng sản phẩm trừu tượng.
* ConcreteFactory (MotifWidgetFactory, PMWidgetFactory): thực hiện các hoạt động để tạo đối tượng sản phẩm cụ thể.
* AbstractProduct (Window, ScrollBar): khai báo một giao diện cho một loại đối tượng sản phẩm.
* ConcreteProduct (MotifWindow, MotifScrollBar)

+ Xác định một đối tượng sản phẩm được tạo bởi nhà máy cụ thể tương ứng.

+ Triển khai giao diện AbstractProduct.

* Client: chỉ sử dụng các giao diện được khai báo bởi các class AbstractFactory và AbstractProduct.

**Ưu điểm:**

* Khuyến khích sự liên kết lỏng lẻo: Mã khách hàng sử dụng Abstract Factory chỉ cần làm việc với các giao diện trừu tượng và không cần biết về các class cụ thể đang được tạo ra. Điều này giúp tách biệt mã khách hàng và chi tiết cài đặt sản phẩm.
* Hỗ trợ việc tạo ra các nhóm object: **Abstract factory design pattern** cho phép tạo ra các object liên quan thuộc cùng một nhóm hoặc họ. Điều này đảm bảo các sản phẩm được tạo ra bởi một xưởng là tương thích và hoạt động tốt cùng nhau.
* Đơn giản hóa việc tạo object: Bằng cách đóng gói logic tạo object trong Abstract Factory và các phiên bản cụ thể của nó, độ phức tạp của việc tạo object được ẩn đi khỏi mã khách hàng. Điều này giúp đơn giản hóa mã khách hàng và tạo ra một thiết kế sạch hơn và dễ bảo trì hơn.
* Dễ dàng thay thế sản phẩm: Vì mã khách hàng hoạt động với các giao diện trừu tượng, việc thay thế một nhóm sản phẩm bằng nhóm khác trở nên dễ dàng hơn.

**Nhược điểm:** Code có thể trở nên phức tạp hơn mức cần thiết, vì rất nhiều giao diện và class mới được giới thiệu cùng với pattern.

**Kiến trúc:**

The Abstract Factory định nghĩa một Factory Method cho mỗi product. Mỗi Phương thức Factory Method đóng gói toán tử mới và các class sản phẩm cụ thể, dành riêng cho nền tảng. Sau đó, mỗi "platform" được mô hình hóa bằng một Factory derived class.

**A diagram of a product

Description automatically generated with medium confidence**

**Template đối với Python:**

"""

Provide an interface for creating families of related or dependent

objects without specifying their concrete classes.

"""

import abc

class AbstractFactory(metaclass=abc.ABCMeta):

    """

    Declare an interface for operations that create abstract product

    objects.

    """

    @abc.abstractmethod

    def create\_product\_a(self):

        pass

    @abc.abstractmethod

    def create\_product\_b(self):

        pass

class ConcreteFactory1(AbstractFactory):

    """

    Implement the operations to create concrete product objects.

    """

    def create\_product\_a(self):

        return ConcreteProductA1()

    def create\_product\_b(self):

        return ConcreteProductB1()

class ConcreteFactory2(AbstractFactory):

    """

    Implement the operations to create concrete product objects.

    """

    def create\_product\_a(self):

        return ConcreteProductA2()

    def create\_product\_b(self):

        return ConcreteProductB2()

class AbstractProductA(metaclass=abc.ABCMeta):

    """

    Declare an interface for a type of product object.

    """

    @abc.abstractmethod

    def interface\_a(self):

        pass

class ConcreteProductA1(AbstractProductA):

    """

    Define a product object to be created by the corresponding concrete

    factory.

    Implement the AbstractProduct interface.

    """

    def interface\_a(self):

        pass

class ConcreteProductA2(AbstractProductA):

    """

    Define a product object to be created by the corresponding concrete

    factory.

    Implement the AbstractProduct interface.

    """

    def interface\_a(self):

        pass

class AbstractProductB(metaclass=abc.ABCMeta):

    """

    Declare an interface for a type of product object.

    """

    @abc.abstractmethod

    def interface\_b(self):

        pass

class ConcreteProductB1(AbstractProductB):

    """

    Define a product object to be created by the corresponding concrete

    factory.

    Implement the AbstractProduct interface.

    """

    def interface\_b(self):

        pass

class ConcreteProductB2(AbstractProductB):

    """

    Define a product object to be created by the corresponding concrete

    factory.

    Implement the AbstractProduct interface.

    """

    def interface\_b(self):

        pass

def main():

    for factory in (ConcreteFactory1(), ConcreteFactory2()):

        product\_a = factory.create\_product\_a()

        product\_b = factory.create\_product\_b()

        product\_a.interface\_a()

        product\_b.interface\_b()

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

    main()

**Code ví dụ**: Đoạn code dưới tạo ra một hệ thống các object liên quan trong 2 họ sản phẩm khác nhau: họ Linux và họ Windows thông qua việc sử dụng Abstract Factory Design Pattern

from abc import ABC, abstractmethod

#abstract base product

class widget(ABC):

    def draw(self):

        pass

#concrete product family 1

class LinuxButton(widget):

    def draw(self):

        print("LinuxButton")

class LinuxMenu(widget):

    def draw(self):

        print("LinuxMenu")

#concrete product family 2

class WindowButton(widget):

    def draw(self):

        print("WindowButton")

class WindowMenu(widget):

    def draw(self):

        print("WindowMenu")

class Factory(ABC):

    @abstractmethod

    def create\_button(self):

        pass

    @abstractmethod

    def create\_menu(self):

        pass

# Concrete factory for Linux

class LinuxFactory(Factory):

    def create\_button(self):

        return LinuxButton()

    def create\_menu(self):

        return LinuxMenu()

# Concrete factory for Windows

class WindowsFactory(Factory):

    def create\_button(self):

        return WindowButton()

    def create\_menu(self):

        return WindowMenu()

class Client:

    def \_\_init\_\_(self, factory):

        self.factory = factory

    def draw(self):

        w = self.factory.create\_button()

        w.draw()

        self.display\_window\_one()

        self.display\_window\_two()

    def display\_window\_one(self):

        w = [self.factory.create\_button(), self.factory.create\_menu()]

        w[0].draw()

        w[1].draw()

    def display\_window\_two(self):

        w = [self.factory.create\_menu(), self.factory.create\_button()]

        w[0].draw()

        w[1].draw()

# Main function

def main():

    factory = None

    # Switch between Linux and Windows factories

    platform = "LINUX"  # Change to "WINDOWS" for Windows factory

    if platform == "LINUX":

        factory = LinuxFactory()

    else:  # WINDOWS

        factory = WindowsFactory()

    client = Client(factory)

    client.draw()

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

    main()

**Giải thích code:**

* Khi chạy đoạn code, tùy theo việc chọn platform LINUX hoặc Windows, nó sẽ tạo ra một Factory tương ứng với LinuxFactory hoặc là WindowsFactory tương ứng với Client.
* Client sẽ sử dụng Factory để tạo các object widget (Button và Menu) của các họ sản phẩm tương ứng. Sau đó thì nó sẽ gọi các phương thức draw() để vẽ chúng.
* Client cũng triển khai hai phương thức display\_window\_one() và display\_window\_two(), trong đó nó tạo ra một cửa sổ giao diện bằng cách kết hợp các object Button và Menu. Mục đích là hiển thị khả năng tương thích giữa các object của cùng một gia đình sản phẩm.
* Với việc sử dụng mẫu Abstract Factory, code cho phép thay thế dễ dàng giữa các gia đình sản phẩm khác nhau (Linux và Windows) chỉ bằng cách thay đổi Factory tương ứng. Điều này giúp tách rời mã khách (Client) và các object cụ thể, đồng thời cho phép mở rộng hệ thống để hỗ trợ thêm các gia đình sản phẩm khác nữa.

### **Builder design pattern**

<https://viblo.asia/p/builder-design-pattern-6J3ZgjwgKmB>

**Giới thiệu:**

* Builder là một mẫu thiết kế (design pattern) thuộc nhóm Creational Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Builder là tách rời quá trình xây dựng (construction) đối tượng phức tạp từ việc biểu diễn nội dung của đối tượng, giúp cho việc xây dựng và tạo đối tượng trở nên linh hoạt và dễ dàng.
* Khi xây dựng một đối tượng phức tạp, có thể có nhiều bước và quy trình phức tạp liên quan. Trong trường hợp này, Builder pattern cho phép chúng ta tạo ra một đối tượng Builder riêng biệt, chịu trách nhiệm xây dựng và cung cấp một giao diện để định nghĩa các bước xây dựng từng phần của đối tượng cuối cùng.
* Pattern này đặc biệt hữu ích khi xử lý các object phức tạp có nhiều tùy chọn hoặc tham số cấu hình.
* Tuần suất sử dụng: trung bình thấp.

**Mục tiêu:** Tách việc xây dựng một đối tượng phức hợp khỏi cách biểu diễn của nó để cùng một quá trình xây dựng có thể tạo ra các cách biểu diễn khác nhau.

**Áp dụng:** Ta sử dụng Builder Pattern khi:

* Thuật toán để tạo một đối tượng phức tạp phải độc lập với các bộ phận tạo nên đối tượng và cách chúng được lắp ráp.
* Quá trình xây dựng phải cho phép các biểu diễn khác nhau cho đối tượng được xây dựng.

**Các thành phần chính:**

* Builder (TextConverter): chỉ định giao diện trừu tượng để tạo các phần của đối tượng Sản phẩm
* ConcreteBuilder (ASCIIConverter, TeXConverter, TextWidgetConverter):

+ Xây dựng và lắp ráp các bộ phận của sản phẩm bằng cách triển khai giao diện Builder.

+ Xác định và theo dõi biểu diễn mà nó tạo ra.

+ Cung cấp giao diện để truy xuất sản phẩm (ví dụ: GetASCIIText, GetTextWidget).

* Director (RTFReader) xây dựng một đối tượng bằng giao diện Builder.
* Product (ASCIIText, TeXText, TextWidget):

+ Đại diện cho đối tượng phức tạp đang được xây dựng. ConcreteBuilder xây dựng biểu diễn bên trong của sản phẩm và xác định quy trình lắp ráp sản phẩm.

+ Bao gồm các class định nghĩa các bộ phận cấu thành, bao gồm các giao diện để lắp ráp các bộ phận thành kết quả cuối cùng.

**Ưu điểm của Builder design pattern**

* Linh hoạt trong việc tạo object: **Builder design pattern** cung cấp tính linh hoạt trong việc tạo object phức tạp với các cấu hình khác nhau. Nó cho phép bạn xây dựng object từng bước một, cung cấp sự kiểm soát chi tiết với quá trình tạo object. Tính linh hoạt này cho phép tạo ra các biến thể object khác nhau mà không làm rối trích xuất hoặc giới thiệu một số lượng lớn tham số.
* Đóng gói logic xây dựng: **Builder design pattern** đóng gói logic xây dựng trong một class xây dựng hoặc giao diện riêng biệt. Sự đóng gói này thúc đẩy việc tổ chức mã và phân tách quan tâm. Chi tiết xây dựng được trừu tượng hoá khỏi mã khách hàng, làm cho mã dễ hiểu và dễ bảo trì hơn.
* Đọc và bảo trì mã: Bằng cách sử dụng **Builder design pattern**, có thể cải thiện tính đọc và khả năng bảo trì mã. Cách tiếp cận từng bước của việc xây dựng object tăng cường sự rõ ràng và khả năng hiểu mã. Ngoài ra, việc thay đổi hoặc bổ sung quy trình xây dựng có thể được thực hiện trong class xây dựng mà không ảnh hưởng đến mã khách hàng, làm cho cơ sở mã dễ bảo trì và mở rộng hơn.
* Độ nhất quán của object: **Builder design pattern** giúp đảm bảo object được xây dựng trong trạng thái nhất quán và hợp lệ. Vì quá trình xây dựng tập trung trong class xây dựng, nó có thể áp dụng quy tắc và ràng buộc để đảm bảo object được khởi tạo đúng cách trước khi sử dụng. Điều này giúp ngăn chặn việc object không được trong trạng thái nhất quán hoặc chưa hoàn chỉnh.
* Dễ dàng tạo object phức tạp: **Builder design pattern** đơn giản hóa việc tạo ra các object phức tạp bằng cách cung cấp một giao diện rõ ràng và trực quan để xây dựng chúng. Nó cho phép bạn chỉ định cấu hình hoặc thuộc tính mong muốn của object bằng cách sử dụng một chuỗi các cuộc gọi phương thức, giúp quá trình tạo object dễ quản lý và ít gặp lỗi hơn.

**Kiến trúc**

**A picture containing text, diagram, font, line

Description automatically generated**

**Template đối với python**

"""

Separate the construction of a complex object from its representation so

that the same construction process can create different representations.

"""

import abc

class Director:

    """

    Construct an object using the Builder interface.

    """

    def \_\_init\_\_(self):

        self.\_builder = None

    def construct(self, builder):

        self.\_builder = builder

        self.\_builder.\_build\_part\_a()

        self.\_builder.\_build\_part\_b()

        self.\_builder.\_build\_part\_c()

class Builder(metaclass=abc.ABCMeta):

    """

    Specify an abstract interface for creating parts of a Product

    object.

    """

    def \_\_init\_\_(self):

        self.product = Product()

    @abc.abstractmethod

    def \_build\_part\_a(self):

        pass

    @abc.abstractmethod

    def \_build\_part\_b(self):

        pass

    @abc.abstractmethod

    def \_build\_part\_c(self):

        pass

class ConcreteBuilder(Builder):

    """

    Construct and assemble parts of the product by implementing the

    Builder interface.

    Define and keep track of the representation it creates.

    Provide an interface for retrieving the product.

    """

    def \_build\_part\_a(self):

        pass

    def \_build\_part\_b(self):

        pass

    def \_build\_part\_c(self):

        pass

class Product:

    """

    Represent the complex object under construction.

    """

    pass

def main():

    concrete\_builder = ConcreteBuilder()

    director = Director()

    director.construct(concrete\_builder)

    product = concrete\_builder.product

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

    main()

**Code ví dụ**: đoạn code dưới cho phép xây dựng các đối tượng các đối tượng Vehicle với các bộ phận khác nhau (frame, engine, wheels, doors) thông qua việc sử dụng Builder Design Pattern.

from abc import ABC, abstractmethod

class Vehicle:

    def \_\_init\_\_(self, vehicle\_type):

        self.vehicle\_type = vehicle\_type

        self.parts = {}

    def set\_part(self, part, value):

        self.parts[part] = value

    def show(self):

        print("---------------------------")

        print("Vehicle Type: {}".format(self.vehicle\_type))

        print(" Frame: {}".format(self.parts["frame"]))

        print(" Engine: {}".format(self.parts["engine"]))

        print(" #Wheels: {}".format(self.parts["wheels"]))

        print(" #Doors: {}".format(self.parts["doors"]))

class VehicleBuilder(ABC):

    def \_\_init\_\_(self):

        self.vehicle = Vehicle("")

    def get\_vehicle(self):

        return self.vehicle

    def build\_frame(self):

        pass

    def build\_engine(self):

        pass

    def build\_wheels(self):

        pass

    def build\_doors(self):

        pass

class MotorCycleBuilder(VehicleBuilder):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def build\_frame(self):

        self.vehicle.set\_part("frame", "MotorCycle Frame")

    def build\_engine(self):

        self.vehicle.set\_part("engine", "500 cc")

    def build\_wheels(self):

        self.vehicle.set\_part("wheels", "2")

    def build\_doors(self):

        self.vehicle.set\_part("doors", "0")

class CarBuilder(VehicleBuilder):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def build\_frame(self):

        self.vehicle.set\_part("frame", "Car Frame")

    def build\_engine(self):

        self.vehicle.set\_part("engine", "2500 cc")

    def build\_wheels(self):

        self.vehicle.set\_part("wheels", "4")

    def build\_doors(self):

        self.vehicle.set\_part("doors", "4")

class ScooterBuilder(VehicleBuilder):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

    def build\_frame(self):

        self.vehicle.set\_part("frame", "Scooter Frame")

    def build\_engine(self):

        self.vehicle.set\_part("engine", "50 cc")

    def build\_wheels(self):

        self.vehicle.set\_part("wheels", "2")

    def build\_doors(self):

        self.vehicle.set\_part("doors", "0")

class Shop:

    def construct(self, vehicle\_builder):

        vehicle\_builder.build\_frame()

        vehicle\_builder.build\_engine()

        vehicle\_builder.build\_wheels()

        vehicle\_builder.build\_doors()

        return vehicle\_builder.get\_vehicle()

def main():

    shop = Shop()

    builder = ScooterBuilder()

    scooter = shop.construct(builder)

    scooter.show()

    builder = CarBuilder()

    car = shop.construct(builder)

    car.show()

    builder = MotorCycleBuilder()

    motorcycle = shop.construct(builder)

    motorcycle.show()

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

    main()

**Giải thích code**

* Đây là một ví dụ về mẫu thiết kế Builder được triển khai trong Python. Mã xác định một tập hợp các class đại diện cho các loại phương tiện khác nhau: Xe, XeBuilder, MotorCycleBuilder, CarBuilder, ScooterBuilder và Cửa hàng.
* Class Vehicle là sản phẩm đang được chế tạo. Nó có một hàm tạo nhận một đối số vehicle\_type và khởi tạo một từ điển các bộ phận trống. Nó cũng có một phương thức set\_part để thiết lập một phần cụ thể của chiếc xe và một phương thức show để hiển thị các chi tiết của chiếc xe.
* Class VehicleBuilder là một class cơ sở trừu tượng (ABC) xác định giao diện để xây dựng một chiếc xe. Nó có các phương pháp trừu tượng để xây dựng các bộ phận khác nhau của phương tiện: build\_frame, build\_engine, build\_wheels và build\_doors. Nó cũng có phương thức get\_vehicle để truy xuất phương tiện đã tạo.
* Các class MotorCycleBuilder, CarBuilder và ScooterBuilder là những bộ tạo cụ thể kế thừa từ VehicleBuilder. Mỗi người xây dựng thực hiện các phương pháp trừu tượng để xây dựng các bộ phận cụ thể của loại phương tiện tương ứng.
* Class Shop là giám đốc điều phối quá trình xây dựng. Nó có một phương thức cấu trúc lấy đối số vehicle\_builder và gọi các phương thức của trình tạo để tạo phương tiện. Sau đó, nó trả lại phương tiện được xây dựng. Chức năng chính tạo một thể hiện của class Shop và sử dụng nó để xây dựng các phương tiện khác nhau bằng cách sử dụng các trình tạo khác nhau. Nó thể hiện tính linh hoạt của mẫu Builder bằng cách tạo ra một chiếc xe tay ga, ô tô và xe máy với các cấu hình khác nhau.
* Khi bạn chạy chức năng chính, nó sẽ xuất ra thông tin chi tiết của từng chiếc xe được chế tạo, hiển thị loại xe, khung, động cơ, số bánh và số cửa.
* Mẫu thiết kế Builder tách biệt việc xây dựng các đối tượng phức tạp khỏi biểu diễn của chúng, cho phép cùng một quy trình xây dựng tạo ra các biểu diễn khác nhau. Nó cung cấp một cách rõ ràng và linh hoạt để xây dựng các đối tượng theo từng bước.

### **Prototype**

**Giới thiệu:**

* Prototype là một design pattern thuộc nhóm Creational Pattern - những mẫu thiết kế cho việc khởi tạo object của class.
* Prototype quy định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này mà không làm cho code phụ thuộc vào các class của chúng.
* Prototype Pattern được dùng khi việc tạo một object tốn nhiều chi phí và thời gian trong khi bạn đã có một object tương tự tồn tại.
* Prototype Pattern cung cấp cơ chế để copy từ object ban đầu sang object mới và thay đổi giá trị một số thuộc tính nếu cần.
* Tần suất sử dụng: trung bình

**Mục tiêu:** Chỉ định các loại đối tượng cần tạo bằng cách sử dụng một phiên bản nguyên mẫu và tạo các đối tượng mới bằng cách sao chép nguyên mẫu này. Xác định một giao diện để tạo một đối tượng, nhưng hãy để các class con quyết định class nào sẽ khởi tạo. Factory Method cho phép một class trì hoãn việc khởi tạo thành các class con.

**Áp dụng:**

**Các thành phần chính:**

* Prototype (Graphic): Khai báo một giao diện để sao chép chính nó.
* ConcretePrototype (Staff, WholeNote, HalfNote): Triển khai một phương thức để sao chép chính nó.
* Client (Công cụ đồ họa): Tạo một đối tượng mới bằng cách yêu cầu nguyên mẫu sao chép chính nó.

**Ưu điểm:**

* Bạn có thể sao chép các đối tượng mà không cần nối với các class cụ thể của chúng.
* Bạn có thể loại bỏ mã khởi tạo lặp đi lặp lại để sao chép các nguyên mẫu dựng sẵn.
* Bạn có thể sản xuất các đối tượng phức tạp thuận tiện hơn.
* Bạn có một giải pháp thay thế cho tính kế thừa khi xử lý các cài đặt trước cấu hình cho các đối tượng phức tạp.

**Nhược điểm:** Nhân bản các đối tượng phức tạp có tham chiếu vòng tròn có thể rất phức tạp.

**Cấu trúc:**

**A picture containing text, diagram, screenshot, font

Description automatically generated**

**Template đối với Python:**

"""

Specify the kinds of objects to create using a prototypical instance,

and create new objects by copying this prototype.

"""

import copy

class Prototype:

    """

    Example class to be copied.

    """

    pass

def main():

    prototype = Prototype()

    prototype\_copy = copy.deepcopy(prototype)

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

    main()

**Ví dụ:** Dưới đây là một ví dụ về việc sử dụng Prototype pattern để tạo các đối tượng sao chép cho các môn thể thao trong Python:

import copy

class Sport:

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

        self.name = name

        self.players = players

    def \_\_str\_\_(self):

        return f"{self.name}: {self.players} players"

    def clone(self, \*\*kwargs):

        clone = copy.copy(self)

        clone.\_\_dict\_\_.update(kwargs)

        return clone

def main():

# Tạo các đối tượng môn thể thao ban đầu

    football = Sport("Football", 11)

    basketball = Sport("Basketball", 5)

    tennis = Sport("Tennis", 2)

    # Sao chép các đối tượng môn thể thao

    football\_clone = football.clone()

    basketball\_clone = basketball.clone()

    tennis\_clone = tennis.clone()

    # Thay đổi số lượng người chơi của các đối tượng đã sao chép

    football\_clone.players = 7

    basketball\_clone.players = 3

    tennis\_clone.players = 1

    # In thông tin các đối tượng

    print(football)         # Kết quả: Football: 11 players

    print(football\_clone)   # Kết quả: Football: 7 players

    print(basketball)       # Kết quả: Basketball: 5 players

    print(basketball\_clone) # Kết quả: Basketball: 3 players

    print(tennis)           # Kết quả: Tennis: 2 players

    print(tennis\_clone)     # Kết quả: Tennis: 1 players

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

    main()

**Giải thích code:**

* Class Sport đại diện cho một môn thể thao. Mỗi đối tượng Sport có hai thuộc tính: name (tên môn thể thao) và players (số lượng người chơi). Phương thức clone được triển khai để sao chép đối tượng Sport và cập nhật các thuộc tính mới nếu cần.
* Trong hàm main, chúng ta tạo ra các đối tượng ban đầu cho các môn thể thao như football, basketball, và tennis. Sau đó, chúng ta sử dụng phương thức clone để sao chép các đối tượng này và lưu trữ trong các biến football\_clone, basketball\_clone, và tennis\_clone.
* Tiếp theo, chúng ta thay đổi giá trị thuộc tính players của các đối tượng đã sao chép để hiển thị các đối tượng có số lượng người chơi khác nhau.
* Cuối cùng, chúng ta in thông tin của các đối tượng ban đầu và các đối tượng đã sao chép để xác nhận rằng số lượng người chơi đã được cập nhật đúng như mong đợi.

### **Singleton**

**Giới thiệu:**

* Singleton là một mẫu thiết kế (design pattern) thuộc nhóm Creational Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Singleton là đảm bảo rằng một class chỉ có duy nhất một thể hiện (instance) duy nhất trong suốt quá trình chạy của ứng dụng.
* Với Singleton, một class được thiết kế sao cho nó chỉ có thể được khởi tạo một lần duy nhất và cung cấp một cách truy cập toàn cục đến thể hiện duy nhất đó. Điều này đảm bảo rằng mọi yêu cầu tạo đối tượng sẽ trả về cùng một thể hiện và đồng thời cho phép các thành phần khác trong hệ thống truy cập và sử dụng thể hiện đó một cách dễ dàng.
* Một cách thông thường để triển khai Singleton là sử dụng một biến tĩnh (static) trong class đó để lưu trữ thể hiện duy nhất và một phương thức tĩnh (static) để truy cập và khởi tạo thể hiện đó. Trong quá trình khởi tạo, class Singleton kiểm tra xem thể hiện đã được tạo chưa. Nếu chưa, nó sẽ tạo mới thể hiện và lưu trữ vào biến tĩnh, sau đó trả về thể hiện đó. Nếu thể hiện đã tồn tại, nó chỉ đơn giản trả về thể hiện đã có.
* Tuần suất sử dụng: cao trung bình.

**Mục tiêu:** Đảm bảo 1 class chỉ có một phiên bản (instance) và cung cấp một điểm truy cập toàn cầu cho nó.

**Áp dụng:** ta sữ dụng Singleton pattern khi

* Phải có chính xác một thể hiện của một class và nó phải có thể truy cập được khách hàng từ một điểm truy cập nổi tiếng.
* Khi nào phiên bản duy nhất nên được mở rộng bằng cách phân class và máy khách sẽ có thể sử dụng một phiên bản mở rộng mà không cần sửa đổi code.

**Các thành phần chính:**

Singleton:

* Định nghĩa một thao tác Phiên bản cho phép khách hàng truy cập phiên bản duy nhất của nó. Phiên bản là một hoạt động của class (nghĩa là một phương thức class trong Smalltalk và một hàm static thành viên trong C++).
* Có thể chịu trách nhiệm tạo phiên bản duy nhất của riêng mình.

**Ưu điểm:**

* Có thể chắc chắn rằng một class chỉ có một thể hiện duy nhất.
* Có được một điểm truy cập toàn cầu cho trường hợp đó.
* Đối tượng singleton chỉ được khởi tạo khi nó được yêu cầu lần đầu tiên.

**Nhược điểm:**

* Vi phạm Nguyên tắc Trách nhiệm Duy nhất. Mô hình giải quyết hai vấn đề tại thời điểm đó.
* Ví dụ, mẫu Singleton có thể che giấu thiết kế xấu khi các thành phần của chương trình biết quá nhiều về nhau.
* Mẫu yêu cầu xử lý đặc biệt trong môi trường đa luồng để nhiều luồng sẽ không tạo một đối tượng đơn lẻ nhiều lần.
* Có thể khó kiểm tra đơn vị mã máy khách của Singleton vì nhiều khung kiểm tra dựa vào tính kế thừa khi tạo các đối tượng giả. Vì phương thức khởi tạo của class singleton là riêng tư và việc ghi đè các phương thức tĩnh là không thể trong hầu hết các ngôn ngữ, bạn sẽ cần nghĩ ra một cách sáng tạo để chế nhạo class singleton. Hoặc chỉ cần không viết các bài kiểm tra. Hoặc không sử dụng mẫu Singleton.

**Cấu trúc:**

**A picture containing text, font, screenshot, line

Description automatically generated**

**Template đối với Python:**

"""

Ensure a class only has one instance, and provide a global point of

access to it.

"""

class Singleton(type):

    """

    Define an Instance operation that lets clients access its unique

    instance.

    """

    def \_\_init\_\_(cls, name, bases, attrs, \*\*kwargs):

        super().\_\_init\_\_(name, bases, attrs)

        cls.\_instance = None

    def \_\_call\_\_(cls, \*args, \*\*kwargs):

        if cls.\_instance is None:

            cls.\_instance = super().\_\_call\_\_(\*args, \*\*kwargs)

        return cls.\_instance

class MyClass(metaclass=Singleton):

    """

    Example class.

    """

    pass

def main():

    m1 = MyClass()

    m2 = MyClass()

    assert m1 is m2

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

    main()

**Code ví dụ:** triển khai mẫu Singleton cho các loại đồ ăn trong Python

class Singleton(type):

    \_instances = {}

    def \_\_call\_\_(cls, \*args, \*\*kwargs):

        if cls not in cls.\_instances:

            cls.\_instances[cls] = super().\_\_call\_\_(\*args, \*\*kwargs)

        return cls.\_instances[cls]

class Food(metaclass=Singleton):

    def \_\_init\_\_(self, name):

        self.name = name

    def \_\_str\_\_(self):

        return self.name

def main():

    burger = Food("Burger")

    pizza = Food("Pizza")

    print(burger)  # Kết quả: Burger

    print(pizza)   # Kết quả: Pizza

    print(burger is pizza)  # Kết quả: True

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

    main()

**Giải thích code:**

* Biến \_instances được sử dụng để lưu trữ các phiên bản của class con đã được tạo ra.
* Phương thức \_\_call\_\_ được ghi đè để kiểm tra xem class con đã có một phiên bản duy nhất nào hay chưa. Nếu chưa có, nó sẽ tạo một phiên bản mới bằng cách gọi super().\_\_call\_\_(\*args, \*\*kwargs) và lưu trữ nó vào biến \_instances. Nếu đã có, nó sẽ trả về phiên bản đã tồn tại.
* Class Food là một ví dụ của class con sử dụng metaclass Singleton. Nó có một thuộc tính name để lưu trữ tên của đồ ăn và các phương thức \_\_init\_\_ và \_\_str\_\_ để khởi tạo và hiển thị tên đồ ăn.
* Trong hàm main(), chúng ta tạo ra hai đối tượng burger và pizza từ class Food. Vì Food sử dụng metaclass Singleton, chỉ có một phiên bản duy nhất của Food được tạo ra. Do đó, cả burger và pizza đều trỏ đến cùng một đối tượng.
* Cuối cùng, chúng ta in ra thông tin về burger và pizza, và kiểm tra xem chúng có cùng tham chiếu hay không. Trong trường hợp này, burger is pizza sẽ trả về True, chỉ ra rằng burger và pizza là cùng một đối tượng Singleton.

## 2.1. Nhóm cấu trúc (Structural Patterns)

(nhóm cấu trúc – 7 mẫu) gồm: *Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy*. Những Design pattern loại này liên quan tới **class** và các thành phần của **object**. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

### **Adapter**

**Giới thiệu:**

* Adapter là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Adapter pattern là chuyển đổi giao diện của một class thành giao diện khác mà các client mong muốn sử dụng.
* Khi xây dựng phần mềm, có thể xảy ra tình huống mà các class hoặc đối tượng đã tồn tại không tương thích với các yêu cầu hoặc giao diện mới. Trong trường hợp này, Adapter pattern cho phép chúng ta tạo ra một class trung gian (adapter) giữa client và đối tượng không tương thích để thích ứng và cung cấp giao diện mà client mong muốn sử dụng.
* Adapter còn được gọi là Wrapper.
* Tần suất sử dụng: cao trung bình.

**Mục tiêu:** Chuyển đổi giao diện của một class thành giao diện của client mong đợi. Adapter cho phép các class làm việc cùng nhau mà không thể do các giao diện không tương thích.

**Áp dụng:** Ta sử dụng Adapter khi:

* Bạn muốn sử dụng một class hiện có và giao diện của nó không khớp với giao diện bạn cần.
* Bạn muốn tạo một class có thể tái sử dụng hợp tác với các class không liên quan hoặc không lường trước được, nghĩa là các class không nhất thiết phải có giao diện tương thích.
* (chỉ bộ điều hợp đối tượng) bạn cần sử dụng một số class con hiện có, nhưng việc điều chỉnh giao diện của chúng bằng cách phân class từng class con là không thực tế. Một bộ điều hợp đối tượng có thể điều chỉnh giao diện của class cha của nó.

**Các thành phần chính:**

* Target (Shape): Xác định các giao diện dành riêng cho domain mà Client sử dụng.
* Client (Drawing Editor): Cộng tác với các objects phù hợp với giao diện Target.
* Adaptee (TextView): Xác định một giao diện hiện có cần điều chỉnh.
* Adapter (TextShape): Điều chỉnh giao diện của Adaptee với giao diện Target.

**Ưu điểm:**

* Đảm bảo nguyên tắc đơn nhiệm (*Single Responsibility Principle*). Bạn có thể tách giao diện hoặc mã chuyển đổi dữ liệu khỏi logic nghiệp vụ chính của chương trình.
* Đảm bảo nguyên tắc Mở/Đóng (*Open/Closed Principle*). Bạn có thể đưa các loại bộ điều hợp mới vào chương trình mà không vi phạm mã máy khách hiện có, miễn là chúng hoạt động với các bộ điều hợp thông qua giao diện máy khách.

**Nhược điểm:** Độ phức tạp tổng thể của code tăng lên vì bạn cần giới thiệu một tập hợp các giao diện và class mới. Đôi khi, việc thay đổi class dịch vụ sao cho phù hợp với phần còn lại của code sẽ đơn giản hơn.

**Cấu trúc:**

Class Adapter sử dụng một loạt các Inheritance để điều chỉnh giao diện này sang giao diện khác:

A picture containing text, diagram, font, line

Description automatically generated

Object Adapter dựa trên thành phần của object:

A picture containing text, diagram, line, font

Description automatically generated

**Template đối với Python:**

"""

Convert the interface of a class into another interface clients expect.

Adapter lets classes work together that couldn't otherwise because of

incompatible interfaces.

"""

import abc

class Target(metaclass=abc.ABCMeta):

    """

    Define the domain-specific interface that Client uses.

    """

    def \_\_init\_\_(self):

        self.\_adaptee = Adaptee()

    @abc.abstractmethod

    def request(self):

        pass

class Adapter(Target):

    """

    Adapt the interface of Adaptee to the Target interface.

    """

    def request(self):

        self.\_adaptee.specific\_request()

class Adaptee:

    """

    Define an existing interface that needs adapting.

    """

    def specific\_request(self):

        pass

def main():

    adapter = Adapter()

    adapter.request()

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

    main()

**Code ví dụ:** chúng ta triển khai mẫu Adapter để phát các file âm thanh với các định dạng khác nhau.

import abc

class MediaPlayer(metaclass=abc.ABCMeta):

    """

    Define the domain-specific interface that Client uses.

    """

    @abc.abstractmethod

    def play(self, audio\_type, file\_name):

        pass

class AdvancedMediaPlayer(metaclass=abc.ABCMeta):

    """

    Define an existing interface that needs adapting.

    """

    @abc.abstractmethod

    def play\_vlc(self, file\_name):

        pass

    @abc.abstractmethod

    def play\_mp4(self, file\_name):

        pass

class VLCPlayer(AdvancedMediaPlayer):

    """

    Class VLCPlayer triển khai Interface AdvancedMediaPlayer để phát các file âm thanh định dạng VLC.

    """

    def play\_vlc(self, file\_name):

        print("Playing VLC file:", file\_name)

    def play\_mp4(self, file\_name):

        pass

class MP4Player(AdvancedMediaPlayer):

    """

    Class MP4Player triển khai Interface AdvancedMediaPlayer để phát các file âm thanh định dạng MP4.

    """

    def play\_vlc(self, file\_name):

        pass

    def play\_mp4(self, file\_name):

        print("Playing MP4 file:", file\_name)

class MediaAdapter(MediaPlayer):

    """

    Class MediaAdapter triển khai Interface MediaPlayer và sử dụng Composition để chuyển đổi giao diện MediaPlayer sang AdvancedMediaPlayer.

    """

    def \_\_init\_\_(self, audio\_type):

        if audio\_type == "vlc":

            self.advanced\_player = VLCPlayer()

        elif audio\_type == "mp4":

            self.advanced\_player = MP4Player()

    def play(self, audio\_type, file\_name):

        if audio\_type == "vlc":

            self.advanced\_player.play\_vlc(file\_name)

        elif audio\_type == "mp4":

            self.advanced\_player.play\_mp4(file\_name)

class AudioPlayer(MediaPlayer):

    """

    Class AudioPlayer triển khai Interface MediaPlayer để phát các file âm thanh. Nếu định dạng file không được hỗ trợ, nó sẽ sử dụng MediaAdapter để chuyển đổi và phát các file âm thanh.

    """

    def play(self, audio\_type, file\_name):

        if audio\_type == "mp3":

            print("Playing MP3 file:", file\_name)

        elif audio\_type == "vlc" or audio\_type == "mp4":

            media\_adapter = MediaAdapter(audio\_type)

            media\_adapter.play(audio\_type, file\_name)

        else:

            print("Unsupported audio format.")

def main():

    audio\_player = AudioPlayer()

    audio\_player.play("mp3", "song.mp3")

    audio\_player.play("vlc", "movie.vlc")

    audio\_player.play("mp4", "video.mp4")

    audio\_player.play("avi", "video.avi")

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

    main()

**Giải thích code:**

* Class MediaPlayer là một interface định nghĩa phương thức play để phát file âm thanh. Interface này được sử dụng bởi Client.
* Class AdvancedMediaPlayer là một interface khác định nghĩa các phương thức play\_vlc và play\_mp4 để phát các file âm thanh định dạng VLC và MP4.
* Class VLCPlayer triển khai interface AdvancedMediaPlayer để phát các file âm thanh định dạng VLC. Phương thức play\_vlc được cài đặt để in ra thông báo "Playing VLC file" kèm theo tên file.
* Class MP4Player triển khai interface AdvancedMediaPlayer để phát các file âm thanh định dạng MP4. Phương thức play\_mp4 được cài đặt để in ra thông báo "Playing MP4 file" kèm theo tên file.
* Class MediaAdapter triển khai interface MediaPlayer và sử dụng Composition để chuyển đổi giao diện của MediaPlayer sang AdvancedMediaPlayer. Trong hàm khởi tạo, class này tạo ra một đối tượng VLCPlayer hoặc MP4Player tương ứng với định dạng âm thanh được yêu cầu. Phương thức play của MediaAdapter sẽ gọi đến phương thức tương ứng của đối tượng VLCPlayer hoặc MP4Player.
* Class AudioPlayer triển khai interface MediaPlayer để phát các file âm thanh. Trong phương thức play, nếu định dạng file là MP3, nó sẽ in ra thông báo "Playing MP3 file". Nếu định dạng file là VLC hoặc MP4, nó sẽ sử dụng MediaAdapter để chuyển đổi và phát file âm thanh tương ứng. Nếu định dạng file không được hỗ trợ, nó sẽ in ra thông báo "Unsupported audio format."
* Trong hàm main(), chúng ta tạo một đối tượng AudioPlayer và gọi phương thức play với các định dạng âm thanh khác nhau như MP3, VLC, MP4 và AVI.

### **Bridge**

**Giới thiệu:**

* Bridge pattern là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Bridge pattern là tách rời một hiện thực (implementation) của một thành phần từ giao diện (abstraction) của nó, để hai thành phần có thể thay đổi độc lập nhau mà không ảnh hưởng đến nhau.
* Bridge pattern được sử dụng khi có một sự phân tách giữa một giao diện và cách thức thực hiện nó. Thay vì kết nối trực tiếp một giao diện với một cách thức thực hiện cụ thể, Bridge pattern giúp chúng ta định nghĩa một cấu trúc có thể thay đổi để kết nối hai phần tách biệt này.
* Brigde Pattern còn được gọi là Handle/Body.
* Tuần suất sử dụng: trung bình.

**Mục tiêu:** Tách rời một sự trừu tượng khỏi việc thực hiện nó để cả hai có thể thay đổi độc lập.

**Áp dụng:** Ta sử dụng Bridge khi:

* Bạn muốn tránh sự ràng buộc lâu dài giữa một sự trừu tượng hóa và sự thực thi của nó. Ví dụ, đây có thể là trường hợp khi triển khai phải được chọn hoặc chuyển đổi trong thời gian chạy.
* Cả phần trừu tượng và phần triển khai của chúng phải được mở rộng bằng cách phân class. Trong trường hợp này, Bridge Pattern cho phép bạn kết hợp các phần trừu tượng và triển khai khác nhau và mở rộng chúng một cách độc lập.
* Những thay đổi trong việc triển khai một sự trừu tượng sẽ không ảnh hưởng đến clients; nghĩa là code của họ không cần phải biên dịch lại.
* (C++) bạn muốn ẩn hoàn toàn việc thực hiện của một sự trừu tượng khỏi các máy clients. Trong C++, biểu diễn của một class được hiển thị trong giao diện class.
* Bạn có nhiều class như đã trình bày trước đó trong sơ đồ Động lực đầu tiên. Hệ thống phân cấp class như vậy chỉ ra nhu cầu chia một đối tượng thành hai phần. Rumbaugh sử dụng thuật ngữ "khái quát hóa lồng nhau" để chỉ các hệ thống phân loại như vậy.
* Bạn muốn chia sẻ một sự thực thi giữa nhiều objects (có thể sử dụng cách đếm tham chiếu), và thực tế này nên được ẩn khỏi máy khách. Một ví dụ đơn giản là class Chuỗi của Coplien, trong đó nhiều đối tượng có thể chia sẻ cùng một biểu diễn chuỗi (StringRep).

**Các thành phần chính:**

* Abstraction (Window):

+ Xác định giao diện của Abstraction

+ Duy trì một tham chiếu đến một object thuộc kiểu người chuyển file

* RefmedAbstraction (IconWindow): Mở rộng giao diện được xác định bởi Abstraction.
* Implementor (WindowImp): định nghĩa giao diện cho việc thực hiện class. Giao diện này không nhất thiết phải tương ứng chính xác với giao diện của Abstraction; thực chất 2 giao diện có thể khác nhau hoàn toàn. Thông thường, giao diện Implementor cung cấp các hoạt động nguyên thủy và giao diện Abstraction xác định các hoạt động cấp cao hơn dựa trên các hoạt động nguyên thủy này.
* Concretelmplementor (XWindowImp, PMWindowImp): triển khai giao diện Implementor và xác định triển khai cụ thể của nó.

**Ưu điểm:**

* Bạn có thể tạo các class và ứng dụng độc lập với nền tảng.
* Code của Client hoạt động với mức độ trừu tượng cao. Nó không được tiếp xúc với các chi tiết nền tảng.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các bản tóm tắt và triển khai mới một cách độc lập với nhau.
* Đảm bảo tính đa nhiệm. Bạn có thể tập trung vào logic cấp cao trong phần trừu tượng hóa và vào các chi tiết nền tảng trong quá trình triển khai.

**Nhược điểm:** Bạn có thể làm cho mã phức tạp hơn bằng cách áp dụng mẫu cho một class có tính gắn kết cao.

**Cấu trúc:**A picture containing diagram, text, sketch, plan

Description automatically generated

**Template cho Python:**

"""

Decouple an abstraction from its implementation so that the two can vary

independently.

"""

import abc

class Abstraction:

    """

    Define the abstraction's interface.

    Maintain a reference to an object of type Implementor.

    """

    def \_\_init\_\_(self, imp):

        self.\_imp = imp

    def operation(self):

        self.\_imp.operation\_imp()

class Implementor(metaclass=abc.ABCMeta):

    """

    Define the interface for implementation classes. This interface

    doesn't have to correspond exactly to Abstraction's interface; in

    fact the two interfaces can be quite different. Typically the

    Implementor interface provides only primitive operations, and

    Abstraction defines higher-level operations based on these

    primitives.

    """

    @abc.abstractmethod

    def operation\_imp(self):

        pass

class ConcreteImplementorA(Implementor):

    """

    Implement the Implementor interface and define its concrete

    implementation.

    """

    def operation\_imp(self):

        pass

class ConcreteImplementorB(Implementor):

    """

    Implement the Implementor interface and define its concrete

    implementation.

    """

    def operation\_imp(self):

        pass

def main():

    concrete\_implementor\_a = ConcreteImplementorA()

    abstraction = Abstraction(concrete\_implementor\_a)

    abstraction.operation()

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

    main()

**Code ví dụ:** Ta dùng Bridge Pattern áp dụng cho hoa và màu sắc của hoa:

import abc

# Abstraction

class Flower(metaclass=abc.ABCMeta):

    def \_\_init\_\_(self, color):

        self.color = color

    @abc.abstractmethod

    def display(self):

        pass

# Refined Abstraction

class Rose(Flower):

    def display(self):

        print(f"A beautiful rose with color {self.color}")

class Lily(Flower):

    def display(self):

        print(f"A lovely lily with color {self.color}")

# Implementor

class Color(metaclass=abc.ABCMeta):

    @abc.abstractmethod

    def apply\_color(self, flower):

        pass

# Concrete Implementor

class RedColor(Color):

    def apply\_color(self, flower):

        print("Applying red color to the flower:")

        flower.display()

class BlueColor(Color):

    def apply\_color(self, flower):

        print("Applying blue color to the flower:")

        flower.display()

def main():

    rose = Rose("Red")

    lily = Lily("White")

    red\_color = RedColor()

    blue\_color = BlueColor()

    red\_color.apply\_color(rose)

    blue\_color.apply\_color(rose)

    red\_color.apply\_color(lily)

    blue\_color.apply\_color(lily)

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

    main()

**Giải thích code:**

* Class Flower là class Abstraction, định nghĩa một giao diện chung cho các loại hoa và chứa một thuộc tính color để đại diện cho màu sắc của hoa.
* Class Rose và Lily là các class Refined Abstraction, mở rộng class Flower và cung cấp triển khai cụ thể cho phương thức display(). Mỗi loại hoa sẽ hiển thị thông tin về loại hoa và màu sắc của nó.
* Class Color là class Implementor, định nghĩa một giao diện chung cho việc áp dụng màu sắc lên hoa.
* Class RedColor và BlueColor là các class Concrete Implementor, triển khai phương thức apply\_color() để áp dụng màu sắc cụ thể cho hoa. Mỗi class màu sắc sẽ hiển thị thông báo về việc áp dụng màu sắc và sau đó gọi phương thức display() của hoa tương ứng.
* Trong hàm main(), chúng ta tạo ra các đối tượng hoa Rose và Lily với màu sắc khác nhau. Sau đó, chúng ta tạo ra các đối tượng màu sắc RedColor và BlueColor. Bằng cách gọi phương thức apply\_color() của các đối tượng màu sắc và truyền vào đối tượng hoa tương ứng, chúng ta áp dụng màu sắc lên hoa và hiển thị thông tin của hoa trên màn hình.
* Kết quả sẽ là mỗi loại hoa sẽ được hiển thị với màu sắc tương ứng trên màn hình, ví dụ: "A beautiful rose with color Red".

### **Composite**

**Giới thiệu:**

* Composite pattern là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Composite pattern là tạo ra một cấu trúc cây phân cấp để đại diện cho cả các đối tượng riêng lẻ và các nhóm đối tượng, đồng thời cho phép chúng được sử dụng một cách thống nhất.
* Composite pattern xây dựng trên nguyên tắc "part-whole", trong đó các đối tượng cá nhân và các nhóm đối tượng đều được coi là các phần của cùng một cấu trúc. Mỗi thành phần trong cấu trúc có thể là một đối tượng riêng lẻ hoặc một nhóm đối tượng.
* Tuần suất sử dụng: Cao trung bình.

**Mục tiêu:** Sắp xếp các đối tượng thành cấu trúc cây để thể hiện cấu trúc phân cấp một phần-toàn bộ. Composite cho phép khách hàng xử lý các đối tượng riêng lẻ và thành phần của các đối tượng một cách thống nhất.

**Áp dụng:** Ta sử dụng Composite khi:

* Bạn muốn biểu diễn phân cấp từng phần của các đối tượng.
* Bạn muốn khách hàng có thể bỏ qua sự khác biệt giữa thành phần của các đối tượng và các đối tượng riêng lẻ. Client sẽ xử lý tất cả các đối tượng trong cấu trúc tổng hợp một cách thống nhất.

**Các thành phần chính:**

* Component (Graphic):

+ Khai báo giao diện cho các đối tượng trong thành phần.

+ Thực hiện hành vi mặc định cho giao diện chung cho tất cả các class, như phù hợp.

+ Khai báo một giao diện để truy cập và quản lý các thành phần con của nó.

+ (tùy chọn) Xác định giao diện để truy cập phần tử cha của thành phần trong cấu trúc đệ quy và triển khai giao diện đó nếu phù hợp.

* Leaf (Rectangle, Line, Text,etc.)

+ Đại diện cho các đối tượng lá trong thành phần. Một chiếc lá không có children (con).

+ Xác định hành vi cho các đối tượng nguyên thủy trong thành phần.

* Composite (Picture):

+ Xác định hành vi cho các thành phần có children.

+ Lưu trữ các thành phần children.

+ Thực hiện các hoạt động liên quan đến children trong giao diện Component.

* Client: thao tác với các đối tượng trong thành phần thông qua giao diện Component.

**Ưu điểm:**

* Bạn có thể làm việc với các cấu trúc cây phức tạp một cách thuận tiện hơn: sử dụng tính đa hình và đệ quy để tạo lợi thế cho bạn.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể đưa các loại phần tử mới vào ứng dụng mà không vi phạm mã hiện có, hiện đang hoạt động với đối tượng cây.

**Nhược điểm:** Có thể khó cung cấp một giao diện chung cho các classes có chức năng khác nhau quá nhiều. Trong một số trường hợp nhất định, bạn phải cần khái quát hóa giao diện thành phần, khiến nó khó hiểu hơn.

**Cấu trúc:**

A diagram of a component

Description automatically generated with low confidence

Một cấu trúc đối tượng Composite điển hình có thể trông như thế này:

A picture containing line, diagram, sketch, white

Description automatically generated

**Template cho Python:**

"""

Compose objects into tree structures to represent part-whole

hierarchies. Composite lets clients treat individual objects and

compositions of objects uniformly.

"""

import abc

class Component(metaclass=abc.ABCMeta):

    """

    Declare the interface for objects in the composition.

    Implement default behavior for the interface common to all classes,

    as appropriate.

    Declare an interface for accessing and managing its child

    components.

    Define an interface for accessing a component's parent in the

    recursive structure, and implement it if that's appropriate

    (optional).

    """

    @abc.abstractmethod

    def operation(self):

        pass

class Composite(Component):

    """

    Define behavior for components having children.

    Store child components.

    Implement child-related operations in the Component interface.

    """

    def \_\_init\_\_(self):

        self.\_children = set()

    def operation(self):

        for child in self.\_children:

            child.operation()

    def add(self, component):

        self.\_children.add(component)

    def remove(self, component):

        self.\_children.discard(component)

class Leaf(Component):

    """

    Represent leaf objects in the composition. A leaf has no children.

    Define behavior for primitive objects in the composition.

    """

    def operation(self):

        pass

def main():

    leaf = Leaf()

    composite = Composite()

    composite.add(leaf)

    composite.operation()

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

    main()

**Code ví dụ**: chúng ta sử dụng Composite Pattern để mô phỏng việc mua đồ trong siêu thị.

import abc

class Component(metaclass=abc.ABCMeta):

    """

    Định nghĩa giao diện chung cho các thành phần trong cấu trúc composite.

    """

    @abc.abstractmethod

    def buy(self):

        pass

class Product(Component):

    """

    Đại diện cho một sản phẩm cụ thể trong siêu thị.

    """

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

        self.name = name

        self.price = price

    def buy(self):

        print(f"Purchased product: {self.name}, Price: {self.price}")

class Category(Component):

    """

    Đại diện cho một danh mục sản phẩm trong siêu thị. Bao gồm nhiều sản phẩm hoặc danh mục con khác.

    """

    def \_\_init\_\_(self, name):

        self.name = name

        self.children = []

    def add(self, component):

        self.children.append(component)

    def remove(self, component):

        self.children.remove(component)

    def buy(self):

        print(f"Purchased category: {self.name}")

        for child in self.children:

            child.buy()

def main():

    # Tạo các sản phẩm

    product\_1 = Product("Milk", 2.5)

    product\_2 = Product("Bread", 1.0)

    product\_3 = Product("Eggs", 3.0)

    # Tạo danh mục sản phẩm nước uống

    drinks\_category = Category("Drinks")

    drinks\_category.add(product\_1)

    drinks\_category.add(product\_2)

    # Tạo danh mục sản phẩm thực phẩm

    groceries\_category = Category("Groceries")

    groceries\_category.add(drinks\_category)

    groceries\_category.add(product\_3)

    # Mua hàng

    groceries\_category.buy()

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

    main()

**Giải thích code:**

* Class Component là class gốc trong cấu trúc Composite, định nghĩa giao diện chung cho cả các sản phẩm và danh mục sản phẩm.
* Class Product là class lá (leaf), đại diện cho một sản phẩm cụ thể trong siêu thị. Nó triển khai phương thức buy() để hiển thị thông tin về sản phẩm đã mua.
* Class Category là class composite, đại diện cho một danh mục sản phẩm trong siêu thị. Nó có thể chứa nhiều sản phẩm hoặc các danh mục con khác. Class này triển khai phương thức add() và remove() để quản lý danh sách các thành phần con và phương thức buy() để hiển thị thông tin về danh mục và mua các thành phần con.
* Trong hàm main(), chúng ta tạo ra các đối tượng sản phẩm product\_1, product\_2, product\_3 và các đối tượng danh mục drinks\_category và groceries\_category. Chúng ta sử dụng phương thức add() để thêm sản phẩm vào danh mục và sau đó gọi phương thức buy() trên groceries\_category để mua tất cả các thành phần con trong danh mục.

### **Decorator**

**Giới thiệu:**

* Decorator pattern là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Decorator pattern là cho phép mở rộng chức năng của một đối tượng mà không cần thay đổi cấu trúc ban đầu của nó.
* Decorator pattern xây dựng trên nguyên tắc "wrapping" (bao bọc), trong đó chúng ta bọc một đối tượng trong một class Decorator để mở rộng hoặc thay đổi chức năng của đối tượng gốc. Bằng cách làm điều này, chúng ta có thể áp dụng các thay đổi mà không làm ảnh hưởng đến các đối tượng khác trong cùng hệ thống.
* Tuần suất sử dụng: Trung bình.

**Mục tiêu:** Đính kèm các trách nhiệm bổ sung cho một đối tượng một cách linh hoạt. Decorators cung cấp một giải pháp thay thế linh hoạt cho phân class để mở rộng chức năng

**Áp dụng:** Ta sử dụng Decorator khi:

* Thêm trách nhiệm cho các đối tượng riêng lẻ một cách linh hoạt và trong suốt, nghĩa là không ảnh hưởng đến các đối tượng khác.
* Đối với những trách nhiệm có thể được rút lại.
* Khi mở rộng bằng phân class là không thực tế. Đôi khi có thể có một số lượng lớn các phần mở rộng độc lập và sẽ tạo ra sự bùng nổ các class con để hỗ trợ mọi sự kết hợp. Hoặc một định nghĩa class có thể bị ẩn hoặc không có sẵn để phân class.

**Các thành phần chính:**

* Component (VisualComponent): Xác định giao diện cho objects có thể có các trách nhiệm được them vào chúng một cách linh hoạt.
* ConcreteComponent (TextView): Xác định một object mà các trách nhiệm bổ sung có thể được gắn vào.
* Decorator: Duy trì tham chiếu đến một object Component và xác định giao diện phù hợp với giao diện của Component.
* ConcreteDecorator (BorderDecorator, ScrollDecorator): Thêm trách nhiệm cho Component.

**Ưu điểm:**

* Bạn có thể mở rộng hành vi của một đối tượng mà không cần tạo một class con mới.
* Bạn có thể thêm hoặc loại bỏ trách nhiệm khỏi một đối tượng trong thời gian chạy.
* Bạn có thể kết hợp một số hành vi bằng cách gói một đối tượng vào nhiều Decorator
* Đảm bảo tính đa nhiệm. Bạn có thể chia một class nguyên khối thực hiện nhiều biến thể có thể có của hành vi thành nhiều class nhỏ hơn.

**Nhược điểm:**

* Thật khó để xóa một trình wrapper cụ thể khỏi wrapper stack.
* Thật khó để triển khai một trình trang trí theo cách mà hành vi của nó không phụ thuộc vào thứ tự trong Decorators stack.
* Mã cấu hình ban đầu của các class có thể trông khá xấu.

**Cấu trúc:**

A picture containing text, diagram, plan, line

Description automatically generated

**Template cho Python:**

"""

Attach additional responsibilities to an object dynamically. Decorators

provide a flexible alternative to subclassing for extending

functionality.

"""

import abc

class Component(metaclass=abc.ABCMeta):

    """

    Define the interface for objects that can have responsibilities

    added to them dynamically.

    """

    @abc.abstractmethod

    def operation(self):

        pass

class Decorator(Component, metaclass=abc.ABCMeta):

    """

    Maintain a reference to a Component object and define an interface

    that conforms to Component's interface.

    """

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

        self.\_component = component

    @abc.abstractmethod

    def operation(self):

        pass

class ConcreteDecoratorA(Decorator):

    """

    Add responsibilities to the component.

    """

    def operation(self):

        # ...

        self.\_component.operation()

        # ...

class ConcreteDecoratorB(Decorator):

    """

    Add responsibilities to the component.

    """

    def operation(self):

        # ...

        self.\_component.operation()

        # ...

class ConcreteComponent(Component):

    """

    Define an object to which additional responsibilities can be

    attached.

    """

    def operation(self):

        pass

def main():

    concrete\_component = ConcreteComponent()

    concrete\_decorator\_a = ConcreteDecoratorA(concrete\_component)

    concrete\_decorator\_b = ConcreteDecoratorB(concrete\_decorator\_a)

    concrete\_decorator\_b.operation()

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

    main()

**Code ví dụ:**

import abc

class Gun(metaclass=abc.ABCMeta):

    """

    Define the base interface for a gun.

    """

    @abc.abstractmethod

    def assemble(self):

        pass

class BaseGun(Gun):

    """

    Define the base implementation of a gun.

    """

    def assemble(self):

        print("Assembling the base gun.")

class GunDecorator(Gun, metaclass=abc.ABCMeta):

    """

    Define the base decorator class that wraps a Gun object.

    """

    def \_\_init\_\_(self, gun):

        self.gun = gun

    def assemble(self):

        self.gun.assemble()

class Scope(GunDecorator):

    """

    Add a scope attachment to the gun.

    """

    def assemble(self):

        super().assemble()

        print("Adding a scope attachment.")

class Silencer(GunDecorator):

    """

    Add a silencer attachment to the gun.

    """

    def assemble(self):

        super().assemble()

        print("Adding a silencer attachment.")

class ExtendedMagazine(GunDecorator):

    """

    Add an extended magazine to the gun.

    """

    def assemble(self):

        super().assemble()

        print("Adding an extended magazine.")

def main():

    # Assemble a base gun

    base\_gun = BaseGun()

    base\_gun.assemble()

    print("")

    # Assemble a gun with scope and silencer

    scoped\_silenced\_gun = Silencer(Scope(BaseGun()))

    scoped\_silenced\_gun.assemble()

    print("")

    # Assemble a gun with extended magazine

    extended\_mag\_gun = ExtendedMagazine(BaseGun())

    extended\_mag\_gun.assemble()

    print("")

    # Assemble a gun with all attachments

    fully\_customized\_gun = ExtendedMagazine(Silencer(Scope(BaseGun())))

    fully\_customized\_gun.assemble()

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

    main()

**Giải thích code:**

* Class Gun là một class trừu tượng cơ bản xác định giao diện cho một súng. Nó khai báo phương thức assemble(), đại diện cho hành động lắp ráp một súng.
* Class BaseGun là một hiện thực cụ thể của giao diện Gun. Nó cung cấp chức năng cơ bản cho việc lắp ráp một súng bằng cách triển khai phương thức assemble().
* Class GunDecorator là một decorator trừu tượng mở rộng giao diện Gun và giữ một tham chiếu đến một đối tượng Gun. Nó triển khai phương thức assemble() bằng cách gọi phương thức của đối tượng Gun được bọc.
* Các decorator cụ thể như Scope, Silencer và ExtendedMagazine kế thừa từ class GunDecorator. Mỗi decorator thêm chức năng cụ thể cho súng bằng cách ghi đè phương thức assemble() và gọi thực thi cơ sở trước khi thêm hành vi riêng.
* Trong hàm main(), một số loại súng được lắp ráp với các kết hợp phụ kiện khác nhau:
* Đầu tiên, tạo và lắp ráp một súng cơ bản sử dụng class BaseGun.
* Tiếp theo, lắp ráp một súng với ống ngắm và củng cố bằng cách bọc đối tượng BaseGun với các decorator Scope và Silencer.
* Sau đó, lắp ráp một súng với một lỗ sứ mở rộng bằng cách sử dụng decorator ExtendedMagazine.
* Cuối cùng, lắp ráp một súng tùy chỉnh hoàn chỉnh bằng cách sử dụng các decorator ExtendedMagazine, Silencer, Scope và BaseGun.

### **Facade**

**Giới thiệu:**

* Facade pattern là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Facade pattern là cung cấp một giao diện đơn giản và thân thiện cho việc tương tác với một hệ thống phức tạp, giúp giảm độ phức tạp và tăng tính module của mã nguồn.
* Khi xây dựng một hệ thống phức tạp, có nhiều class và các phần tử phụ thuộc vào nhau. Facade pattern đóng vai trò như một class trung gian (facade) giữa khách hàng và các class phức tạp trong hệ thống. Nó cung cấp một giao diện đơn giản và thân thiện để khách hàng có thể tương tác với hệ thống một cách dễ dàng.
* Tuần suất sử dụng: Cao.

**Mục tiêu:** Cung cấp một giao diện thống nhất cho một tập hợp các giao diện trong một hệ thống con. Facade xác định giao diện cấp cao hơn giúp hệ thống con dễ sử dụng hơn.

**Áp dụng:** Ta sử dụng Facade khi:

* Bạn muốn cung cấp một giao diện đơn giản cho một hệ thống con phức tạp. Các hệ thống con thường trở nên phức tạp hơn khi chúng phát triển. Hầu hết các Pattern khi được áp dụng sẽ dẫn đến các class ngày càng nhỏ hơn. Điều này làm cho hệ thống con có thể tái sử dụng nhiều hơn và dễ tùy chỉnh hơn, nhưng nó cũng trở nên khó sử dụng hơn đối với các máy Client không cần tùy chỉnh nó. Façade có thể cung cấp chế độ xem mặc định đơn giản của hệ thống con đủ tốt cho hầu hết Clients. Chỉ những Clients cần nhiều khả năng tùy chỉnh hơn mới cần nhìn xa hơn Facade.
* Có nhiều sự phụ thuộc giữa các Clients và các class thực thi của một sự trừu tượng hóa. Giới thiệu một Facade để tách hệ thống con khỏi Clients và các hệ thống con khác, do đó thúc đẩy tính di động và tính độc lập của hệ thống con.
* Bạn muốn xếp class các hệ thống con của mình. Sử dụng một Façade để xác định một điểm vào cho từng cấp độ hệ thống con. Nếu các hệ thống phụ phụ thuộc, thì bạn có thể đơn giản hóa các phụ thuộc giữa chúng bằng cách làm cho chúng giao tiếp với nhau chỉ thông qua các Facade của chúng.

**Các thành phần chính:**

* Facade (Compiler):

+ Biết class hệ thống con nào chịu trách nhiệm cho một yêu cầu.

+ Ủy quyền các yêu cầu của máy khách cho các đối tượng hệ thống con thích hợp.

* Subsystem classes (Scanner, Parser, ProgramNode, v.v.)

+ Thực hiện chức năng hệ thống con.

+ Xử lý công việc do đối tượng Facade giao.

+ Không có kiến thức về Facade; nghĩa là, họ không có tài liệu tham khảo nào về nó.

**Ưu điểm:** Bạn có thể tách code của mình khỏi sự phức tạp của một hệ thống con.

**Nhược điểm:** Facade có thể trở thành một god object được kết hợp với tất cả các class của ứng dụng.

**Cấu trúc:**

A picture containing diagram, line, parallel, plan

Description automatically generated

**Template cho Python:**

"""

Provide a unified interface to a set of interfaces in a subsystem.

Facade defines a higher-level interface that makes the subsystem easier

to use.

"""

class Facade:

    """

    Know which subsystem classes are responsible for a request.

    Delegate client requests to appropriate subsystem objects.

    """

    def \_\_init\_\_(self):

        self.\_subsystem\_1 = Subsystem1()

        self.\_subsystem\_2 = Subsystem2()

    def operation(self):

        self.\_subsystem\_1.operation1()

        self.\_subsystem\_1.operation2()

        self.\_subsystem\_2.operation1()

        self.\_subsystem\_2.operation2()

class Subsystem1:

    """

    Implement subsystem functionality.

    Handle work assigned by the Facade object.

    Have no knowledge of the facade; that is, they keep no references to

    it.

    """

    def operation1(self):

        pass

    def operation2(self):

        pass

class Subsystem2:

    """

    Implement subsystem functionality.

    Handle work assigned by the Facade object.

    Have no knowledge of the facade; that is, they keep no references to

    it.

    """

    def operation1(self):

        pass

    def operation2(self):

        pass

def main():

    facade = Facade()

    facade.operation()

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

    main()

**Code ví dụ:**

class ProductCatalog:

    """

    The ProductCatalog class represents the product catalog or inventory.

    """

    def \_\_init\_\_(self):

        self.products = {

            'iphone': 999,

            'macbook': 1999,

            'ipad': 799

        }

    def get\_product\_price(self, product):

        return self.products.get(product)

class ShoppingCart:

    """

    The ShoppingCart class represents the shopping cart.

    """

    def \_\_init\_\_(self):

        self.items = []

    def add\_item(self, product):

        self.items.append(product)

    def remove\_item(self, product):

        self.items.remove(product)

    def get\_items(self):

        return self.items

    def calculate\_total(self):

        total = 0

        for item in self.items:

            total += item.get\_price()

        return total

class PaymentGateway:

    """

    The PaymentGateway class represents the payment gateway.

    """

    def process\_payment(self, amount):

        print(f"Processing payment of ${amount}")

        # Additional payment processing logic goes here

class OrderFacade:

    """

    The OrderFacade class provides a simplified interface for ordering products online through a phone.

    It hides the complexity of interacting with the catalog, shopping cart, and payment gateway.

    """

    def \_\_init\_\_(self):

        self.catalog = ProductCatalog()

        self.cart = ShoppingCart()

        self.payment\_gateway = PaymentGateway()

    def add\_to\_cart(self, product):

        price = self.catalog.get\_product\_price(product)

        if price:

            item = Product(product, price)

            self.cart.add\_item(item)

            print(f"Added {product} to the cart.")

        else:

            print(f"{product} is not available.")

    def remove\_from\_cart(self, product):

        item = next((item for item in self.cart.get\_items() if item.get\_name() == product), None)

        if item:

            self.cart.remove\_item(item)

            print(f"Removed {product} from the cart.")

        else:

            print(f"{product} is not in the cart.")

    def checkout(self):

        total = self.cart.calculate\_total()

        print(f"Total amount to be paid: ${total}")

        self.payment\_gateway.process\_payment(total)

        print("Order placed successfully!")

class Product:

    """

    The Product class represents a product.

    """

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

        self.name = name

        self.price = price

    def get\_name(self):

        return self.name

    def get\_price(self):

        return self.price

# Client code

def main():

    facade = OrderFacade()

    facade.add\_to\_cart('iphone')

    facade.add\_to\_cart('macbook')

    facade.remove\_from\_cart('macbook')

    facade.add\_to\_cart('ipad')

    facade.checkout()

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

    main()

**Giải thích code:**

* ProductCatalog (Danh mục sản phẩm): Class này đại diện cho danh mục sản phẩm hoặc kho hàng. Nó lưu trữ thông tin về các sản phẩm và giá của chúng.
* ShoppingCart (Giỏ hàng): Class này đại diện cho giỏ hàng. Nó cho phép thêm sản phẩm vào giỏ hàng, xóa sản phẩm khỏi giỏ hàng, lấy danh sách các sản phẩm trong giỏ hàng và tính tổng số tiền cần thanh toán.
* PaymentGateway (Cổng thanh toán): Class này đại diện cho cổng thanh toán. Trong ví dụ này, nó chỉ đơn giản là in ra thông báo xử lý thanh toán. Trong thực tế, bạn sẽ thêm mã xử lý thanh toán thực tế vào đây.
* OrderFacade (Giao diện đặt hàng): Class này cung cấp một giao diện đơn giản để đặt hàng sản phẩm trực tuyến thông qua điện thoại. Nó che giấu sự phức tạp của tương tác với danh mục sản phẩm, giỏ hàng và cổng thanh toán. Nó cung cấp các phương thức để thêm sản phẩm vào giỏ hàng, xóa sản phẩm khỏi giỏ hàng và thanh toán đơn hàng.
* Product (Sản phẩm): Class này đại diện cho một sản phẩm cụ thể. Nó lưu trữ thông tin về tên và giá của sản phẩm.
* Trong hàm main(), chúng ta tạo một đối tượng OrderFacade và sử dụng giao diện của nó để thực hiện quá trình đặt hàng.

### **Flyweight**

**Giới thiệu:**

* Flyweight pattern là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Flyweight pattern là tối ưu hóa việc sử dụng bộ nhớ bằng cách chia sẻ các đối tượng có tính chất giống nhau.
* Khi một ứng dụng phải xử lý hàng ngàn hoặc hàng triệu đối tượng có tính chất tương tự và có thể được chia sẻ, việc tạo một đối tượng mới cho mỗi trường hợp có thể dẫn đến sự lãng phí bộ nhớ. Flyweight pattern giải quyết vấn đề này bằng cách tách riêng phần tương tự và phần riêng của đối tượng. Phần tương tự được chia sẻ bởi các đối tượng khác nhau, trong khi phần riêng được lưu trữ riêng biệt cho từng đối tượng.
* Tuần suất sử dụng: Thấp

**Mục tiêu:** Sử dụng chia sẻ để hỗ trợ số lượng lớn các đối tượng chi tiết một cách hiệu quả.

**Áp dụng:** Hiệu quả của mẫu Flyweight phụ thuộc rất nhiều vào cách thức và vị trí nó được sử dụng. Áp dụng mẫu Flyweight khi tất cả những điều sau đây đều đúng:

* Một ứng dụng sử dụng một số lượng lớn các đối tượng.
* Chi phí lưu trữ cao do số lượng đối tượng quá lớn.
* Hầu hết trạng thái đối tượng có thể được thực hiện bên ngoài.
* Nhiều nhóm đối tượng có thể được thay thế bằng tương đối ít đối tượng dùng chung sau khi trạng thái bên ngoài bị loại bỏ.
* Ứng dụng không phụ thuộc vào nhận dạng đối tượng. Vì các đối tượng flyweight có thể được chia sẻ, nên các kiểm tra nhận dạng sẽ trả về true cho các đối tượng khác biệt về mặt khái niệm.

**Các thành phần chính:**

* Flyweight (Glyph): Tuyên bố một giao diện thông qua đó các Flyweight có thể nhận và hoạt động ở trạng thái bên ngoài.
* ConcreteFlyweight (Nhân vật): Triển khai giao diện Flyweight và thêm dung lượng lưu trữ cho trạng thái nội tại, nếu có. Đối tượng ConcreteFlyweight phải có thể chia sẻ được. Bất kỳ trạng thái nào nó lưu trữ phải là nội tại; nghĩa là nó phải độc lập với bối cảnh của đối tượng ConcreteFlyweight.
* UnsharedConcreteFlyweight (Hàng, Cột): Không phải tất cả các subclasses Flyweight đều cần được chia sẻ. Giao diện Flyweight cho phép chia sẻ; nó không thực thi nó. Các đối tượng UnsharedConcreteFlyweight thường có các đối tượng ConcreteFlyweight khi còn nhỏ ở một mức nào đó trong cấu trúc đối tượng flyweight (như các class Hàng và Cột có).
* FlyweightFactory:

+ Tạo và quản lý các đối tượng Flyweight.

+ Đảm bảo rằng Flyweight được chia sẻ đúng cách. Khi một Client yêu cầu một flyweight, các đối tượng FlyweightFactory sẽ cung cấp một phiên bản hiện có hoặc tạo một phiên bản nếu không tồn tại.

* Clients:

+ Duy trì tham chiếu đến (các) Flyweight.

+ Tính toán hoặc lưu trữ trạng thái bên ngoài của (các) Flyweight.

**Ưu điểm:** Bạn có thể tiết kiệm rất nhiều RAM, giả sử chương trình của bạn có rất nhiều đối tượng giống nhau.

**Nhược điểm:**

* Bạn có thể đang giao dịch RAM theo chu kỳ CPU khi một số dữ liệu ngữ cảnh cần được tính toán lại mỗi khi ai đó gọi một phương pháp Flyweight.
* Mã trở nên phức tạp hơn nhiều. Các thành viên mới trong nhóm sẽ luôn thắc mắc tại sao trạng thái của một thực thể lại bị tách ra theo cách như vậy.

**Cấu trúc:**

A picture containing text, diagram, plan, screenshot

Description automatically generated

Biểu đồ tiếp theo thể hiện cách các Flyweight chia sẻ với nhau:

A picture containing sketch, diagram, drawing, plan

Description automatically generated

**Template cho Python:**

"""

Use sharing to support large numbers of fine-grained objects

efficiently.

"""

import abc

class FlyweightFactory:

    """

    Create and manage flyweight objects.

    Ensure that flyweights are shared properly. When a client requests a

    flyweight, the FlyweightFactory object supplies an existing instance

    or creates one, if none exists.

    """

    def \_\_init\_\_(self):

        self.\_flyweights = {}

    def get\_flyweight(self, key):

        try:

            flyweight = self.\_flyweights[key]

        except KeyError:

            flyweight = ConcreteFlyweight()

            self.\_flyweights[key] = flyweight

        return flyweight

class Flyweight(metaclass=abc.ABCMeta):

    """

    Declare an interface through which flyweights can receive and act on

    extrinsic state.

    """

    def \_\_init\_\_(self):

        self.intrinsic\_state = None

    @abc.abstractmethod

    def operation(self, extrinsic\_state):

        pass

class ConcreteFlyweight(Flyweight):

    """

    Implement the Flyweight interface and add storage for intrinsic

    state, if any. A ConcreteFlyweight object must be sharable. Any

    state it stores must be intrinsic; that is, it must be independent

    of the ConcreteFlyweight object's context.

    """

    def operation(self, extrinsic\_state):

        pass

def main():

    flyweight\_factory = FlyweightFactory()

    concrete\_flyweight = flyweight\_factory.get\_flyweight("key")

    concrete\_flyweight.operation(None)

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

    main()

**Code ví dụ:**

import abc

class Milk:

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

        self.name = name

        self.price = price

    def describe(self):

        print(f"Milk Type: {self.name}, Price: ${self.price}")

class MilkFlyweightFactory:

    milk\_types = {}

    @staticmethod

    def get\_milk\_type(name, price):

        if name not in MilkFlyweightFactory.milk\_types:

            milk\_type = Milk(name, price)

            MilkFlyweightFactory.milk\_types[name] = milk\_type

        return MilkFlyweightFactory.milk\_types[name]

def main():

    milk\_list = [

        ("Full Cream", 2.5),

        ("Skimmed", 2.0),

        ("Soy", 3.0),

        ("Full Cream", 2.5)

    ]

    flyweight\_factory = MilkFlyweightFactory()

    for milk\_info in milk\_list:

        name, price = milk\_info

        milk\_type = flyweight\_factory.get\_milk\_type(name, price)

        milk\_type.describe()

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

    main()

**Giải thích code:**

* Milk: Đây là class đại diện cho đối tượng Milk. Nó có các thuộc tính như tên (name) và giá (price), và phương thức describe() để mô tả thông tin về loại sữa.
* MilkFlyweightFactory: Đây là class xưởng (factory) Flyweight cho các đối tượng Milk. Nó chứa một từ điển (dictionary) milk\_types để lưu trữ các đối tượng Milk đã được tạo. Phương thức get\_milk\_type(name, price) được sử dụng để truy xuất một đối tượng Milk đã được chia sẻ từ từ điển. Nếu đối tượng chưa tồn tại, nó sẽ tạo mới đối tượng Milk và lưu nó vào từ điển.
* Trong hàm main(), chúng ta tạo một danh sách milk\_list chứa thông tin về loại sữa gồm tên và giá. Sau đó, chúng ta tạo một đối tượng MilkFlyweightFactory. Trong vòng lặp, chúng ta lặp qua danh sách sữa và sử dụng MilkFlyweightFactory để lấy đối tượng Milk. Nếu đối tượng đã tồn tại, nó sẽ được lấy từ xưởng (factory), ngược lại nó sẽ được tạo mới và lưu vào xưởng. Sau đó, chúng ta gọi phương thức describe() trên đối tượng Milk để mô tả thông tin về loại sữa.

### **Proxy**

**Giới thiệu:**

* Proxy pattern là một mẫu thiết kế (design pattern) thuộc nhóm Structural Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Proxy pattern là cung cấp một đại diện (proxy) cho một đối tượng khác và kiểm soát truy cập vào đối tượng đó.
* Proxy pattern được sử dụng khi chúng ta muốn kiểm soát hoặc bổ sung các hoạt động cho một đối tượng khác mà không muốn thay đổi cấu trúc của nó. Proxy pattern tạo ra một class trung gian (proxy) với cùng giao diện như đối tượng gốc, cho phép chúng ta thực hiện các hoạt động bổ sung trước hoặc sau khi gọi đối tượng gốc.
* Tuần suất sử dụng: Cao trung bình.
* Proxy Pattern còn được gọi là Surrogate (trình thay thế)

**Mục tiêu:** Cung cấp một Surrogate hoặc trình giữ chỗ cho một đối tượng khác để kiểm soát quyền truy cập vào nó.

**Áp dụng:** Proxy được áp dụng bất cứ khi nào có nhu cầu tham chiếu linh hoạt hoặc tinh vi hơn tới một đối tượng hơn là một con trỏ đơn giản. Dưới đây là một số tình huống phổ biến trong đó mẫu Proxy được áp dụng:

* Một proxy từ xa cung cấp một đại diện cục bộ cho một đối tượng trong một không gian địa chỉ khác. NEXTSTEP [Add94] sử dụng class NXProxy cho mục đích này. Coplien [Cop92]gọi loại ủy quyền này là "Đại sứ".
* Một proxy ảo tạo các đối tượng đắt tiền theo yêu cầu. ImageProxy được mô tả trong Motivation là một ví dụ về aproxy như vậy.
* Proxy bảo vệ kiểm soát quyền truy cập vào đối tượng ban đầu. Proxy bảo vệ rất hữu ích khi các đối tượng nên có các quyền truy cập khác nhau. Ví dụ: KernelProxies trong hệ điều hành Choices [CIRM93] cung cấp quyền truy cập được bảo vệ vào các đối tượng của hệ điều hành.
* Tham chiếu thông minh là sự thay thế cho con trỏ trống thực hiện các hành động bổ sung khi một đối tượng được truy cập. điển hình bao gồm:

+ Đếm số lượng tham chiếu đến đối tượng thực để nó có thể tự động được giải phóng khi không còn tham chiếu nào nữa (còn gọi là con trỏ thông minh [Ede92]).

+ Tải một đối tượng liên tục vào bộ nhớ khi nó được tham chiếu lần đầu tiên.

+ Kiểm tra xem đối tượng thực đã bị khóa trước khi nó được truy cập để đảm bảo rằng không đối tượng nào khác có thể thay đổi nó.

**Các thành phần chính:**

Đại diện (ImageProxy):

* Duy trì một tham chiếu cho phép Proxy truy cập vào real Subject. Proxy có thể tham chiếu đến một Subject nếu giao diện RealSubject và Subject giống nhau.
* Cung cấp một giao diện giống hệt với Subject để Proxy có thể thay thế cho real Subject.
* Kiểm soát quyền truy cập vào real Subject và có thể chịu trách nhiệm tạo và xóa nó.
* Các trách nhiệm khác phụ thuộc vào loại proxy:

*+ Remote proxies* chịu trách nhiệm mã hóa yêu cầu và các đối số của nó và gửi yêu cầu được mã hóa tới real Subject trong một không gian địa chỉ khác.

*+ Vitural proxies* có thể lưu trữ thông tin bổ sung về real Subject để họ có thể trì hoãn việc truy cập nó. Ví dụ: ImageProxy từ Motivation lưu trữ phạm vi của hình ảnh thực.

*+ Protection proxies* kiểm tra xem người gọi có quyền truy cập cần thiết để thực hiện yêu cầu hay không.

**Ưu điểm:**

* Bạn có thể kiểm soát đối tượng dịch vụ mà không cần Clients biết về nó.
* Bạn có thể quản lý vòng đời của đối tượng dịch vụ khi Clients không quan tâm đến nó.
* Proxy hoạt động ngay cả khi đối tượng dịch vụ chưa sẵn sàng hoặc không khả dụng.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu proxy mới mà không cần thay đổi dịch vụ hoặc Client.

**Nhược điểm:**

* Mã có thể trở nên phức tạp hơn vì bạn cần giới thiệu nhiều class mới.
* Phản hồi từ dịch vụ có thể bị trì hoãn.

**Cấu trúc:**

A picture containing text, diagram, line, plan

Description automatically generated

Đây là sơ đồ đối tượng có thể có của cấu trúc proxy trong thời gian chạy:

A close-up of a box

Description automatically generated with low confidence

**Template cho Python:**

"""

Provide a surrogate or placeholder for another object to control access

to it or add other responsibilities.

"""

import abc

class Subject(metaclass=abc.ABCMeta):

    """

    Define the common interface for RealSubject and Proxy so that a

    Proxy can be used anywhere a RealSubject is expected.

    """

    @abc.abstractmethod

    def request(self):

        pass

class Proxy(Subject):

    """

    Maintain a reference that lets the proxy access the real subject.

    Provide an interface identical to Subject's.

    """

    def \_\_init\_\_(self, real\_subject):

        self.\_real\_subject = real\_subject

    def request(self):

        # ...

        self.\_real\_subject.request()

        # ...

class RealSubject(Subject):

    """

    Define the real object that the proxy represents.

    """

    def request(self):

        pass

def main():

    real\_subject = RealSubject()

    proxy = Proxy(real\_subject)

    proxy.request()

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

    main()

**Code ví dụ:**

import abc

class PaymentService(metaclass=abc.ABCMeta):

    """

    Define the interface for the payment service.

    """

    @abc.abstractmethod

    def pay(self, amount):

        pass

class RealPaymentService(PaymentService):

    """

    The actual implementation of the payment service.

    """

    def pay(self, amount):

        print(f"Paying ${amount}.")

class PaymentProxy(PaymentService):

    """

    The proxy class that controls access to the payment service.

    """

    def \_\_init\_\_(self, payment\_service):

        self.payment\_service = payment\_service

    def pay(self, amount):

        self.\_perform\_additional\_checks(amount)

        self.payment\_service.pay(amount)

        self.\_log\_payment(amount)

    def \_perform\_additional\_checks(self, amount):

        # Perform additional checks or validations here

        print(f"Performing additional checks for payment of ${amount}.")

    def \_log\_payment(self, amount):

        # Log the payment details

        print(f"Payment of ${amount} logged successfully.")

def main():

    # Create an instance of the actual payment service

    real\_payment\_service = RealPaymentService()

    # Create a proxy for the payment service

    payment\_proxy = PaymentProxy(real\_payment\_service)

    # Use the proxy to make a payment

    payment\_proxy.pay(100)

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

    main()

**Giải thích code:**

* PaymentService là một class cơ sở trừu tượng định nghĩa giao diện cho dịch vụ thanh toán. Nó khai báo một phương thức pay(), đại diện cho hoạt động thanh toán.
* RealPaymentService là một class cụ thể thực hiện dịch vụ thanh toán. Nó kế thừa từ PaymentService và cung cấp triển khai thực tế cho phương thức pay(). Trong ví dụ này, nó đơn giản chỉ in ra số tiền thanh toán.
* PaymentProxy là class proxy (đại diện) điều khiển quyền truy cập vào dịch vụ thanh toán. Nó cũng kế thừa từ PaymentService để đảm bảo tuân thủ cùng giao diện. Class proxy giữ một tham chiếu đến dịch vụ thanh toán thực sự và chuyển tiếp hoạt động thanh toán cho nó.
* Trong phương thức pay() của proxy, nó trước tiên thực hiện các kiểm tra hoặc xác thực bổ sung đặc biệt cho quy trình thanh toán. Ở đây, nó in một thông báo cho biết đang tiến hành kiểm tra bổ sung.
* Tiếp theo, proxy gọi phương thức pay() trên dịch vụ thanh toán thực sự, truyền đối số về số tiền nhận được từ proxy.
* Sau khi hoạt động thanh toán được thực thi, proxy ghi lại chi tiết thanh toán bằng cách gọi phương thức riêng \_log\_payment().
* Trong hàm main(), đoạn code mô tả cách sử dụng proxy. Nó tạo một phiên bản của dịch vụ thanh toán thực tế (RealPaymentService) và sau đó tạo một proxy cho dịch vụ thanh toán (PaymentProxy), cung cấp dịch vụ thanh toán thực sự như một đối số.
* Cuối cùng, proxy được sử dụng để thực hiện thanh toán bằng cách gọi phương thức pay() trên thể hiện proxy. Proxy thực hiện các kiểm tra bổ sung, chuyển tiếp thanh toán cho dịch vụ thanh toán thực sự và ghi lại chi tiết thanh toán.

## 2.3. Nhóm Behavioral (nhóm hành vi/ tương tác)

(nhóm tương tác/ hành vi – 11 mẫu) gồm: *Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor*. Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các **object** với nhau.

### **Chain of Responsibility**

**Giới thiệu:**

* Chain of Responsibility pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Chain of Responsibility pattern là xây dựng một chuỗi các đối tượng xử lý (handlers) và cho phép các yêu cầu được chuyển tiếp từ handler này sang handler khác trong chuỗi cho đến khi có một handler có thể xử lý yêu cầu đó.
* Khi chúng ta có một số đối tượng có khả năng xử lý các yêu cầu nhưng không biết trước đối tượng nào sẽ xử lý yêu cầu cụ thể, Chain of Responsibility pattern giúp xây dựng một chuỗi các đối tượng xử lý. Khi một yêu cầu được gửi đến, nó sẽ được chuyển tiếp qua từng đối tượng xử lý trong chuỗi cho đến khi có một đối tượng xử lý có thể xử lý yêu cầu đó.
* Tuần suất sử dụng: Trung bình thấp.

**Mục tiêu:** Tránh ghép nối người gửi yêu cầu với người nhận của nó bằng cách cho nhiều đối tượng cơ hội xử lý yêu cầu. Xâu chuỗi các đối tượng nhận và chuyển yêu cầu dọc theo chuỗi cho đến khi một đối tượng xử lý nó.

**Áp dụng:** Ta sử dụng Chain of Responsibility Pattern khi:

* Nhiều đối tượng có thể xử lý một yêu cầu và trình xử lý không được biết trước. Trình xử lý phải được xác định tự động.
* Bạn muốn gửi yêu cầu tới một trong số các đối tượng mà không chỉ định rõ ràng người nhận.
* Tập hợp các đối tượng có thể xử lý yêu cầu phải được chỉ định động.

**Các thành phần chính:**

* Handler (HelpHandler):

+ Định nghĩa một giao diện để xử lí các yêu cầu.

+ (Tùy chọn) thực hiện liên kết kế tiếp.

* ConcreteHandler (PrintButton, PrintDialog):

+ Xử lí các yêu cầu mà nó chịu trách nhiệm.

+ Có thể truy cập vào người kế nhiệm của nó.

+ nếu ConcreteHandler có thể xử lý yêu cầu, nó sẽ làm như vậy; mặt khác, nó chuyển tiếp yêu cầu tới người kế nhiệm của nó.

* Client: triển khai yêu cầu ConcreteHandler object trên chuỗi.

**Ưu điểm:**

* Bạn có thể kiểm soát thứ tự xử lý yêu cầu.
* Đảm bảo tính đa nhiệm. Bạn có thể tách các class triệu gọi các thao tác ra khỏi các class thực hiện các thao tác.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu trình xử lý mới vào ứng dụng mà không vi phạm mã máy khách hiện có.

**Nhược điểm:** Một số yêu cầu có thể không được xử lý.

**Cấu trúc:**

A picture containing text, diagram, line, font

Description automatically generated

Một cấu trúc đối tượng điển hình có thể trông như thế này:A picture containing text, screenshot, line, font

Description automatically generated

**Template cho Python:**

"""

Avoid coupling the sender of a request to its receiver by giving

more than one object a chance to handle the request. Chain the receiving

objects and pass the request along the chain until an object handles it.

"""

import abc

class Handler(metaclass=abc.ABCMeta):

    """

    Define an interface for handling requests.

    Implement the successor link.

    """

    def \_\_init\_\_(self, successor=None):

        self.\_successor = successor

    @abc.abstractmethod

    def handle\_request(self):

        pass

class ConcreteHandler1(Handler):

    """

    Handle request, otherwise forward it to the successor.

    """

    def handle\_request(self):

        if True:  # if can\_handle:

            pass

        elif self.\_successor is not None:

            self.\_successor.handle\_request()

class ConcreteHandler2(Handler):

    """

    Handle request, otherwise forward it to the successor.

    """

    def handle\_request(self):

        if False:  # if can\_handle:

            pass

        elif self.\_successor is not None:

            self.\_successor.handle\_request()

def main():

    concrete\_handler\_1 = ConcreteHandler1()

    concrete\_handler\_2 = ConcreteHandler2(concrete\_handler\_1)

    concrete\_handler\_2.handle\_request()

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

    main()

**Code ví dụ:**

import abc

class DeliveryHandler(metaclass=abc.ABCMeta):

    """

    The abstract base class for delivery handlers.

    """

    def \_\_init\_\_(self, next\_handler=None):

        self.next\_handler = next\_handler

    def set\_next\_handler(self, handler):

        self.next\_handler = handler

    @abc.abstractmethod

    def handle\_delivery(self, package):

        pass

class DeliveryProcessor(DeliveryHandler):

    """

    A concrete delivery handler that processes the delivery request.

    """

    def handle\_delivery(self, package):

        if package.is\_fragile:

            print("Package is fragile. Applying special handling.")

            # Perform special handling for fragile packages

        else:

            print("Delivering the package.")

class DeliveryNotification(DeliveryHandler):

    """

    A concrete delivery handler that sends delivery notifications.

    """

    def handle\_delivery(self, package):

        print("Sending delivery notification.")

        # Send delivery notification to the recipient

class DeliveryTracking(DeliveryHandler):

    """

    A concrete delivery handler that tracks the delivery progress.

    """

    def handle\_delivery(self, package):

        print("Tracking the delivery progress.")

        # Track the package's delivery progress

def main():

    # Create the delivery handlers

    delivery\_processor = DeliveryProcessor()

    delivery\_notification = DeliveryNotification()

    delivery\_tracking = DeliveryTracking()

    # Set up the chain of responsibility

    delivery\_processor.set\_next\_handler(delivery\_notification)

    delivery\_notification.set\_next\_handler(delivery\_tracking)

    # Simulate a delivery request

    package = Package(is\_fragile=True)

    # Start the delivery process

    delivery\_processor.handle\_delivery(package)

class Package: #not affect the chain

    """

    Represents a package to be delivered.

    """

    def \_\_init\_\_(self, is\_fragile=False):

        self.is\_fragile = is\_fragile

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

    main()

**Giải thích code:**

* DeliveryHandler là class abstract base class (class trừu tượng) cho các xử lý viên giao hàng. Nó định nghĩa phương thức handle\_delivery() mà các xử lý viên con cần triển khai.
* DeliveryProcessor, DeliveryNotification, và DeliveryTracking là các class xử lý viên cụ thể (concrete handler) thực hiện các nhiệm vụ cụ thể trong quy trình giao hàng. Mỗi class con triển khai phương thức handle\_delivery() để xử lý gói hàng được chuyển đến nó. Các class xử lý viên này có thể được kết nối thành một chuỗi xử lý thông qua phương thức set\_next\_handler() để chuyển gói hàng qua các bước xử lý.
* Trong hàm main(), chúng ta tạo ra các đối tượng xử lý viên và thiết lập chuỗi xử lý bằng cách sử dụng phương thức set\_next\_handler().
* Class Package đại diện cho một gói hàng cần được giao. Trong ví dụ này, chúng ta chỉ sử dụng thuộc tính is\_fragile để chỉ ra xem gói hàng có tính fragile hay không. Thuộc tính này được sử dụng trong DeliveryProcessor để quyết định xử lý đặc biệt cho gói hàng dễ vỡ.
* Cuối cùng, trong hàm main(), chúng ta tạo một đối tượng Package và bắt đầu quy trình giao hàng bằng cách gọi delivery\_processor.handle\_delivery(package). Quy trình giao hàng sẽ chuyển gói hàng qua các xử lý viên theo chuỗi đã thiết lập, từ DeliveryProcessor đến DeliveryNotification và DeliveryTracking, mỗi xử lý viên sẽ thực hiện nhiệm vụ của mình trên gói hàng.

### **Command**

**Giới thiệu:**

* Command pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Command pattern là tách rời người gửi yêu cầu (client) và người nhận yêu cầu (receiver) bằng cách sử dụng một đối tượng trung gian (command).
* Trong Command pattern, một yêu cầu được biểu diễn dưới dạng một đối tượng command, trong đó chứa thông tin cần thiết để thực hiện yêu cầu đó. Đối tượng command này sẽ đóng gói yêu cầu và các tham số tương ứng, sau đó được gửi tới một người nhận yêu cầu (receiver) để thực hiện hành động tương ứng.
* Tuần suất sử dụng: Cao trung bình.
* Command Pattern còn được gọi là Action, Transaction.

**Mục tiêu:** Đóng gói một yêu cầu dưới dạng một đối tượng, do đó cho phép bạn tham số hóa các ứng dụng client với các yêu cầu, hàng đợi hoặc yêu cầu nhật ký khác nhau và hỗ trợ các thao tác không thể hoàn tác.

**Áp dụng:** Sử dụng Command Pattern khi bạn muốn

* Tham số hóa các đối tượng bằng một biểu mẫu hành động, như các đối tượng Menultem đã làm ở trên. Bạn có thể diễn đạt tham số hóa như vậy bằng ngôn ngữ thủ tục với hàm **callback**, nghĩa là, một hàm được đăng ký ở đâu đó để được gọi sau này. Commands là một sự thay thế hướng đối tượng cho các cuộc gọi lại.
* Chỉ định, xếp hàng và thực hiện các yêu cầu tại các thời điểm khác nhau. Một đối tượng Command có thể có thời gian tồn tại độc lập với yêu cầu ban đầu. Nếu người nhận yêu cầu có thể được biểu diễn theo cách không phụ thuộc vào không gian địa chỉ, thì bạn có thể chuyển đối tượng lệnh cho yêu cầu sang một quy trình khác và thực hiện yêu cầu ở đó.
* Hỗ trợ hoàn tác. Thao tác Execute của Commands có thể lưu trữ trạng thái để đảo ngược các hiệu ứng của nó trong chính lệnh đó. Giao diện Command phải có một thao tác Unexecute được thêm vào để đảo ngược tác động của lệnh gọi Execute trước đó. Các lệnh đã thực thi được lưu trữ trong danh sách lịch sử. Hoàn tác và làm lại ở cấp độ không giới hạn đạt được bằng cách duyệt qua danh sách này ngược và xuôi gọi Unexecute và Execute tương ứng.
* Hỗ trợ ghi lại các thay đổi để chúng có thể được áp dụng lại trong trường hợp hệ thống gặp sự cố. Bằng cách tăng cường giao diện Command với các hoạt động tải và lưu trữ, bạn có thể giữ nhật ký thay đổi liên tục. Khôi phục sau sự cố liên quan đến việc tải lại các lệnh đã ghi từ đĩa và thực hiện lại chúng bằng thao tác Execute.
* Cấu trúc một hệ thống xung quanh các hoạt động cấp cao được xây dựng trên các hoạt động nguyên thủy. Cấu trúc như vậy là phổ biến trong các hệ thống thông tin hỗ trợ các giao dịch. Một giao dịch đóng gói một tập hợp các thay đổi đối với dữ liệu. Command Pattern cung cấp một cách để mô hình hóa các giao dịch. Commands có một giao diện chung, cho phép bạn gọi tất cả giao dịch theo cùng một cách. Pattern cũng giúp dễ dàng mở rộng hệ thống với các giao dịch mới.

**Các thành phần chính:**

* Command: khai báo một giao diện để thực hiện một thao tác.
* ConcreteCommand (PasteCommand, OpenCommand):

+ Xác định một ràng buộc giữa một đối tượng Receiver và một hành động.

+ Thực hiện Execute bằng cách gọi (các) thao tác tương ứng trên Bộ thu.

* Client (Application): Tạo một đối tượng ConcreteCommand và đặt bộ thu của nó.
* Invoker (Menultem): Yêu cầu lệnh thực hiện yêu cầu.
* Receiver (Document, Application): Biết thực hiện các thao tác liên quan đến việc thực hiện một yêu cầu. Bất kỳ class nào cũng có thể đóng vai trò là Receiver.

**Ưu điểm:**

* Đảm bảo tính đa nhiệm. Bạn có thể tách các class gọi các thao tác ra khỏi các class thực hiện các thao tác này.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể đưa các lệnh mới vào ứng dụng mà không vi phạm mã máy khách hiện có.
* Bạn có thể thực hiện hoàn tác/làm lại.
* Bạn có thể triển khai thực thi các hoạt động bị trì hoãn.
* Bạn có thể tập hợp một tập hợp các lệnh đơn giản thành một lệnh phức tạp.

**Nhược điểm:** Code có thể trở nên phức tạp hơn vì bạn đang giới thiệu một class hoàn toàn mới giữa senders và receivers.

**Cấu trúc:**A picture containing text, diagram, line, screenshot

Description automatically generated

**Template cho Python:**

"""

Encapsulate a request as an object, thereby letting you parameterize

clients with different requests, queue or log requests, and support

undoable operations.

"""

import abc

class Invoker:

    """

    Ask the command to carry out the request.

    """

    def \_\_init\_\_(self):

        self.\_commands = []

    def store\_command(self, command):

        self.\_commands.append(command)

    def execute\_commands(self):

        for command in self.\_commands:

            command.execute()

class Command(metaclass=abc.ABCMeta):

    """

    Declare an interface for executing an operation.

    """

    def \_\_init\_\_(self, receiver):

        self.\_receiver = receiver

    @abc.abstractmethod

    def execute(self):

        pass

class ConcreteCommand(Command):

    """

    Define a binding between a Receiver object and an action.

    Implement Execute by invoking the corresponding operation(s) on

    Receiver.

    """

    def execute(self):

        self.\_receiver.action()

class Receiver:

    """

    Know how to perform the operations associated with carrying out a

    request. Any class may serve as a Receiver.

    """

    def action(self):

        pass

def main():

    receiver = Receiver()

    concrete\_command = ConcreteCommand(receiver)

    invoker = Invoker()

    invoker.store\_command(concrete\_command)

    invoker.execute\_commands()

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

    main()

**Code ví dụ:**

import abc

class OrderCommand(metaclass=abc.ABCMeta):

    """

    The abstract base class for order commands.

    """

    @abc.abstractmethod

    def execute(self):

        pass

class TakeOrderCommand(OrderCommand):

    """

    A concrete command that represents taking an order.

    """

    def \_\_init\_\_(self, chef, order):

        self.chef = chef

        self.order = order

    def execute(self):

        self.chef.take\_order(self.order)

class CookCommand(OrderCommand):

    """

    A concrete command that represents cooking a food item.

    """

    def \_\_init\_\_(self, chef, food\_item):

        self.chef = chef

        self.food\_item = food\_item

    def execute(self):

        self.chef.cook(self.food\_item)

class Chef:

    """

    The receiver class that executes the commands.

    """

    def take\_order(self, order):

        print(f"Order taken: {order}")

    def cook(self, food\_item):

        print(f"Cooking: {food\_item}")

class Waiter:

    """

    The invoker class that triggers the commands.

    """

    def \_\_init\_\_(self):

        self.commands = []

    def place\_order(self, command):

        self.commands.append(command)

        command.execute()

    def serve\_orders(self):

        for command in self.commands:

            if isinstance(command, CookCommand):

                print("Serving the cooked food.")

            else:

                print("Order served.")

        self.commands = []

def main():

    chef = Chef()

    waiter = Waiter()

    # Take orders

    order1 = "Pizza"

    order2 = "Burger"

    take\_order\_command1 = TakeOrderCommand(chef, order1)

    take\_order\_command2 = TakeOrderCommand(chef, order2)

    waiter.place\_order(take\_order\_command1)

    waiter.place\_order(take\_order\_command2)

    # Cook food

    food\_item1 = "Pizza"

    food\_item2 = "Burger"

    cook\_command1 = CookCommand(chef, food\_item1)

    cook\_command2 = CookCommand(chef, food\_item2)

    waiter.place\_order(cook\_command1)

    waiter.place\_order(cook\_command2)

    # Serve orders

    waiter.serve\_orders()

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

    main()

**Giải thích code:**

* OrderCommand là một class cơ sở trừu tượng định nghĩa giao diện cho các lệnh đặt hàng. Nó bao gồm một phương thức trừu tượng execute(), mà các class lệnh cụ thể phải triển khai.
* TakeOrderCommand là một class lệnh cụ thể đại diện cho việc đặt một món hàng. Nó nhận vào một đối tượng Chef và một món hàng trong hàm khởi tạo và triển khai phương thức execute() bằng cách gọi phương thức take\_order() của đối tượng Chef.
* CookCommand là một class lệnh cụ thể khác đại diện cho việc nấu một món ăn. Nó nhận vào một đối tượng Chef và một món ăn trong hàm khởi tạo và triển khai phương thức execute() bằng cách gọi phương thức cook() của đối tượng Chef.
* Chef là class nhận lệnh thực thi các lệnh. Nó định nghĩa các phương thức take\_order() và cook(), mô phỏng việc nhận đơn hàng và nấu món ăn tương ứng.
* Waiter là class gọi lệnh, chịu trách nhiệm kích hoạt các lệnh. Nó lưu trữ danh sách các lệnh và bao gồm các phương thức place\_order() và serve\_orders(). Phương thức place\_order() thêm một lệnh vào danh sách và thực thi nó, trong khi phương thức serve\_orders() duyệt qua danh sách các lệnh và phục vụ đơn hàng tương ứng.
* Trong hàm main(), chúng ta tạo một đối tượng Chef và một đối tượng Waiter. Sau đó, chúng ta thực hiện các bước sau:
* Đặt các đơn hàng bằng cách tạo các lệnh TakeOrderCommand tương ứng và gọi phương thức place\_order() của Waiter.
* Nấu món ăn bằng cách tạo các lệnh CookCommand tương ứng và gọi lại phương thức place\_order() của Waiter.
* Phục vụ đơn hàng bằng cách gọi phương thức serve\_orders() của Waiter, và tùy thuộc vào loại lệnh, thông báo về việc phục vụ hoặc phục vụ món ăn đã nấu.
* Mục đích của mẫu Command là tách biệt người yêu cầu thực hiện một thao tác với đối tượng thực hiện thao tác đó. Trong trường hợp này, người yêu cầu là class Waiter, đối tượng thực hiện là class Chef, và các lệnh được đại diện bởi các class con của OrderCommand như TakeOrderCommand và CookCommand.

### **Interpreter**

**Giới thiệu:**

* Interpreter pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Interpreter pattern là định nghĩa một ngôn ngữ đại diện cho các biểu diễn và quy tắc xử lý cho một vấn đề cụ thể. Nó cho phép chúng ta giải quyết các vấn đề liên quan đến ngôn ngữ một cách dễ dàng và linh hoạt.
* Interpreter pattern hoạt động dựa trên việc chuyển đổi một câu lệnh hoặc biểu thức từ ngôn ngữ tự nhiên sang một dạng trung gian, sau đó thực hiện quy tắc xử lý trên dạng trung gian đó. Đối tượng Interpreter được sử dụng để đại diện cho các quy tắc cụ thể trong ngôn ngữ.
* Tuần suất sử dụng: Thấp.

**Mục tiêu:** Cho trước một ngôn ngữ, hãy xác định một biểu diễn cho ngữ pháp của ngôn ngữ đó cùng với một trình thông dịch sử dụng biểu diễn đó để diễn giải các câu trong ngôn ngữ đó.

**Áp dụng:** Sử dụng Interpreter Pattern khi có ngôn ngữ để thông dịch và bạn có thể biểu diễn các câu lệnh trong ngôn ngữ dưới dạng cây cú pháp trừu tượng. Interpreter Pattern hoạt động tốt nhất khi:

* Ngữ pháp đơn giản. Đối với các ngữ pháp phức tạp, hệ thống phân cấp class cho ngữ pháp trở nên lớn và không thể quản lý được. Các công cụ như trình tạo trình phân tích cú pháp là một giải pháp thay thế tốt hơn trong những trường hợp như vậy. Họ có thể diễn giải các biểu thức mà không cần xây dựng các cây cú pháp trừu tượng, điều này có thể tiết kiệm không gian và có thể cả thời gian.
* Hiệu quả không phải là một mối quan tâm quan trọng. Các trình thông dịch hiệu quả nhất thường không được thực hiện bằng cách thông dịch trực tiếp các cây phân tích cú pháp mà bằng cách dịch chúng sang một dạng khác trước tiên. Ví dụ, các biểu thức chính quy thường được chuyển thành các máy trạng thái. Nhưng ngay cả khi đó, trình dịch có thể được triển khai bằng mẫu Phiên dịch, vì vậy mẫu này vẫn được áp dụng.

**Các thành phần chính:**

* AbstractExpression (RegularExpression): tuyên bố một hoạt động Thông dịch trừu tượng phổ biến cho các nút trong cây cú pháp trừu tượng.
* TerminalExpression (LiteralExpression):

+ Thực hiện thao tác Thông dịch được liên kết với các ký hiệu đầu cuối trong ngữ pháp.

+ Một ví dụ được yêu cầu cho mọi ký hiệu đầu cuối trong một câu.

* NonterminalExpression (AlternationExpression, RepetitionExpression, SequenceExpressions)

+ Một class như vậy là bắt buộc đối với mọi quy tắc R ::= R­1R2…Rn trong ngữ pháp.

+ Duy trì các biến đối tượng kiểu AbstractExpression cho từng ký hiệu R1 đến Rn.

+ Thực hiện thao tác Interpret cho các ký hiệu không đầu cuối trong ngữ pháp. Interpret thường gọi chính nó theo cách đệ quy trên các biến đại diện cho R1 đến Rn.

* Context: Chứa thông tin toàn cầu đối Interpreter.
* Client:

+ Xây dựng (hoặc được cung cấp) một cây cú pháp trừu tượng đại diện cho một câu cụ thể trong ngôn ngữ mà ngữ pháp xác định. Cây cú pháp trừu tượng được tập hợp từ các thể hiện của các class NonterminalExpression và TerminalExpression classes.

+ Gọi hoạt động Interpret.

**Ưu điểm:**

* Linh hoạt: Interpreter Pattern cho phép thêm mới các hoạt động hoặc quy tắc vào ngôn ngữ mà không cần chỉnh sửa mã hiện có. Điều này tạo ra tính linh hoạt cao và dễ mở rộng.
* Tính mô-đun: Pattern này khuyến khích việc tiếp cận mô-đun trong việc xử lý ngôn ngữ bằng cách chia nhỏ các quy tắc ngôn ngữ phức tạp thành các thành phần nhỏ hơn, dễ quản lý hơn. Mỗi thành phần có thể được triển khai dưới dạng biểu thức riêng biệt, giúp dễ hiểu và bảo trì mã nguồn.
* Tùy chỉnh: Interpreter Pattern cho phép khách hàng tự định nghĩa ngôn ngữ đặc thù của riêng mình bằng cách chỉ định cú pháp và ý nghĩa của ngôn ngữ. Điều này cho phép khách hàng tùy chỉnh ngôn ngữ theo nhu cầu và yêu cầu cụ thể của họ.
* Tính tái sử dụng: Pattern này khuyến khích việc tái sử dụng các class biểu thức đã có sẵn. Sau khi định nghĩa biểu thức, chúng có thể được kết hợp và tái sử dụng trong nhiều ngữ cảnh khác nhau, giảm việc trùng lặp mã nguồn và cải thiện khả năng bảo trì mã nguồn.

**Nhược điểm:**

* Phức tạp: Interpreter Pattern có thể tạo ra độ phức tạp, đặc biệt đối với các ngôn ngữ có quy tắc ngữ pháp phức tạp. Quản lý tương tác giữa các biểu thức khác nhau và đảm bảo sự diễn dịch chính xác của ngôn ngữ có thể gặp khó khăn và yêu cầu thiết kế và triển khai cẩn thận.
* Hiệu năng chậm: Việc diễn dịch một ngôn ngữ có thể tạo ra overhead hiệu năng so với các phương pháp khác, như biên dịch ngôn ngữ thành mã máy. Quá trình diễn dịch có thể chậm hơn, đặc biệt đối với các biểu thức lớn và phức tạp.
* Hạn chế tối ưu hóa: Ngôn ngữ được diễn dịch thông thường có.

**Cấu trúc:** **A picture containing text, diagram, font, line

Description automatically generated**

**Template cho Python:**

"""

Define a represention for a grammar of the given language along with an

interpreter that uses the representation to interpret sentences in the

language.

"""

import abc

class AbstractExpression(metaclass=abc.ABCMeta):

    """

    Declare an abstract Interpret operation that is common to all nodes

    in the abstract syntax tree.

    """

    @abc.abstractmethod

    def interpret(self):

        pass

class NonterminalExpression(AbstractExpression):

    """

    Implement an Interpret operation for nonterminal symbols in the grammar.

    """

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

        self.\_expression = expression

    def interpret(self):

        self.\_expression.interpret()

class TerminalExpression(AbstractExpression):

    """

    Implement an Interpret operation associated with terminal symbols in

    the grammar.

    """

    def interpret(self):

        pass

def main():

    abstract\_syntax\_tree = NonterminalExpression(TerminalExpression())

    abstract\_syntax\_tree.interpret()

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

    main()

**Code ví dụ:**

import abc

class AbstractExpression(metaclass=abc.ABCMeta):

    """

    Declare an abstract Interpret operation that is common to all nodes

    in the abstract syntax tree.

    """

    @abc.abstractmethod

    def interpret(self):

        pass

class NoteExpression(AbstractExpression):

    """

    Implement an Interpret operation for note symbols in the music grammar.

    """

    def \_\_init\_\_(self, note):

        self.note = note

    def interpret(self):

        if self.note == "C":

            return "C là nốt đô - Một nốt nhạc căn bản"

        elif self.note == "D":

            return "D là nốt rê - Một nốt nhạc căn bản"

        elif self.note == "E":

            return "E là nốt mi - Một nốt nhạc căn bản"

        elif self.note == "F":

            return "F là nốt fa - Một nốt nhạc căn bản"

        elif self.note == "G":

            return "G là nốt sol - Một nốt nhạc căn bản"

        elif self.note == "A":

            return "A là nốt la - Một nốt nhạc căn bản"

        elif self.note == "B":

            return "B là nốt si - Một nốt nhạc căn bản"

class MusicInterpreter:

    """

    The client class that interprets musical expressions.

    """

    def \_\_init\_\_(self):

        self.expression = None

    def interpret(self, musical\_expression):

        parts = musical\_expression.split()

        for part in parts:

            if part.isalpha():

                self.expression = NoteExpression(part)

                print(self.expression.interpret())

def main():

    music\_interpreter = MusicInterpreter()

    musical\_expression = "C D E F G A B"

    music\_interpreter.interpret(musical\_expression)

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

    main()

**Giải thích code:**

* Class AbstractExpression là class trừu tượng đại diện cho biểu diễn chung của các nút trong cây cú pháp trừu tượng. Nó khai báo một phương thức trừu tượng interpret().
* Class NoteExpression là class cụ thể thực hiện phép giải thích cho các ký hiệu nốt nhạc trong ngữ pháp âm nhạc. Nó lưu trữ ký hiệu nốt nhạc và triển khai phương thức interpret() để trả về ý nghĩa tương ứng với ký hiệu nốt nhạc.
* Class MusicInterpreter là class khách hàng, nó thực hiện việc giải thích các biểu diễn âm nhạc. Class này có một thuộc tính expression để lưu trữ biểu diễn hiện tại. Phương thức interpret() nhận vào một biểu diễn âm nhạc dưới dạng chuỗi, tách thành các phần riêng biệt và kiểm tra xem mỗi phần có phải là ký tự chữ hay không. Nếu là ký tự chữ, nó tạo một đối tượng NoteExpression và gọi phương thức interpret() để hiển thị ý nghĩa tương ứng với ký hiệu nốt nhạc.
* Hàm main() tạo một đối tượng MusicInterpreter và gọi phương thức interpret() với biểu diễn âm nhạc là "C D E F G A B".

### **Iterator**

**Giới thiệu:**

* Iterator pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Iterator pattern là cung cấp một giao diện chung để truy cập tuần tự các phần tử của một tập hợp (collection) mà không tiết lộ cấu trúc bên trong của tập hợp đó.
* Iterator pattern cho phép duyệt qua từng phần tử của một tập hợp một cách tuần tự, mà không cần quan tâm đến cấu trúc nội bộ của tập hợp đó. Nó tách rời việc duyệt qua tập hợp và cấu trúc dữ liệu của tập hợp, giúp tăng tính linh hoạt và tái sử dụng trong việc truy cập dữ liệu.
* Tuần suất sử dụng: Cao.
* Iterator Pattern còn được gọi là Cursor.

**Mục tiêu:** Cung cấp một cách để truy cập các phần tử của một đối tượng tổng hợp một cách tuần tự mà không để lộ biểu diễn cơ bản của nó.

**Áp dụng:** Sử dụng Iterator Pattern:

* Để truy cập nội dung của một đối tượng tổng hợp mà không để lộ biểu diễn bên trong của nó.
* Để hỗ trợ nhiều lần duyệt các đối tượng tổng hợp.
* Để cung cấp một giao diện thống nhất để duyệt qua các cấu trúc tổng hợp khác nhau (nghĩa là hỗ trợ phép lặp đa hình).

**Các thành phần chính:**

* Iterator: định nghĩa một giao diện để truy cập và duyệt qua các phần tử.
* Concretelterator:

+ Thực hiện giao diện Iterator.

+ Theo dõi vị trí hiện tại trong đường truyền của tổng hợp.

* Aggregate: Định nghĩa một giao diện để tạo một đối tượng Iterator.
* ConcreteAggregate: Thực hiện giao diện tạo Iterator để trả về một thể hiện của Concretelterator thích hợp.

**Ưu điểm:**

* Đảm bảo tính đa nhiệm. Bạn có thể dọn sạch code của Client và Collections bằng cách trích xuất các thuật toán truyền tải cồng kềnh thành các class riêng biệt.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể triển khai các loại collections và iterators mới, đồng thời chuyển chúng vào mã hiện có mà không vi phạm bất kỳ điều gì.
* Bạn có thể lặp song song trên cùng một collection vì mỗi iterator object chứa trạng thái iteration của chính nó.
* Vì lý do tương tự, bạn có thể trì hoãn iterator và tiếp tục khi cần.

**Nhược điểm:**

* Việc áp dụng Pattern có thể là quá mức cần thiết nếu ứng dụng của bạn chỉ hoạt động với các collections đơn giản.
* Việc sử dụng một iterator có thể kém hiệu quả hơn so với việc xem trực tiếp các phần tử của một số bộ sưu tập chuyên biệt.

**Cấu trúc:** A picture containing text, diagram, line, screenshot

Description automatically generated

**Template cho Python:**

"""

Provide a way to access the elements of an aggregate objects equentially

without exposing its underlying representation.

"""

import collections.abc

class ConcreteAggregate(collections.abc.Iterable):

    """

    Implement the Iterator creation interface to return an instance of

    the proper ConcreteIterator.

    """

    def \_\_init\_\_(self):

        self.\_data = None

    def \_\_iter\_\_(self):

        return ConcreteIterator(self)

class ConcreteIterator(collections.abc.Iterator):

    """

    Implement the Iterator interface.

    """

    def \_\_init\_\_(self, concrete\_aggregate):

        self.\_concrete\_aggregate = concrete\_aggregate

    def \_\_next\_\_(self):

        if True:  # if no\_elements\_to\_traverse:

            raise StopIteration

        return None  # return element

def main():

    concrete\_aggregate = ConcreteAggregate()

    for \_ in concrete\_aggregate:

        pass

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

    main()

**Code ví dụ:**

import collections.abc

class HoChiMinhCityTour(collections.abc.Iterable):

    """

    Implement the Iterator creation interface to return an instance of

    the proper ConcreteIterator.

    """

    def \_\_init\_\_(self):

        self.places = []

    def add\_place(self, place):

        self.places.append(place)

    def \_\_iter\_\_(self):

        return HoChiMinhCityIterator(self.places)

class HoChiMinhCityIterator(collections.abc.Iterator):

    """

    Implement the Iterator interface.

    """

    def \_\_init\_\_(self, places):

        self.places = places

        self.current\_index = 0

    def \_\_next\_\_(self):

        if self.current\_index < len(self.places):

            place = self.places[self.current\_index]

            self.current\_index += 1

            return place

        else:

            raise StopIteration

def main():

    hcm\_city\_tour = HoChiMinhCityTour()

    hcm\_city\_tour.add\_place("Ben Thanh Market")

    hcm\_city\_tour.add\_place("War Remnants Museum")

    hcm\_city\_tour.add\_place("Notre-Dame Cathedral Basilica of Saigon")

    hcm\_city\_tour.add\_place("Cu Chi Tunnels")

    print("Exploring Ho Chi Minh City:")

    for place in hcm\_city\_tour:

        print(place)

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

    main()

**Giải thích code:**

* Class HoChiMinhCityTour triển khai giao diện collections.abc.Iterable để đảm bảo rằng nó là một đối tượng có thể lặp qua (iterable). Trong class này, chúng ta có một danh sách các địa điểm (places) và phương thức add\_place để thêm địa điểm vào danh sách. Phương thức \_\_iter\_\_ được triển khai để trả về một đối tượng HoChiMinhCityIterator mới.
* Class HoChiMinhCityIterator triển khai giao diện collections.abc.Iterator. Class này có một danh sách các địa điểm và biến current\_index để theo dõi chỉ số hiện tại của địa điểm đang được truy cập. Phương thức \_\_next\_\_ được triển khai để trả về địa điểm tiếp theo trong danh sách. Nếu đã đi qua tất cả các địa điểm, nó sẽ raise một ngoại lệ StopIteration để kết thúc vòng lặp.
* Trong hàm main, chúng ta tạo một tour tham quan TP Hồ Chí Minh (hcm\_city\_tour) và thêm các địa điểm vào. Sau đó, chúng ta sử dụng vòng lặp for để duyệt qua các địa điểm trong tour và in chúng ra màn hình.

### **Mediator**

**Giới thiệu:**

* Mediator pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Mediator pattern là giảm sự phụ thuộc giữa các đối tượng trong một hệ thống phức tạp bằng cách tạo ra một đối tượng trung gian (mediator) để điều phối thông tin và tương tác giữa các đối tượng khác nhau.
* Trong mô hình Mediator, các đối tượng không trực tiếp tương tác với nhau, mà thay vào đó tương tác thông qua đối tượng mediator. Mediator có nhiệm vụ điều phối và xử lý thông tin từ các đối tượng và quyết định cách tương tác giữa chúng. Điều này giúp giảm sự phụ thuộc giữa các đối tượng và giữ cho chúng độc lập với nhau.
* Tuần suất sử dụng: Trung bình Thấp.

**Mục tiêu:** Xác định một đối tượng gói gọn cách một tập hợp các đối tượng tương tác. Mediator thúc đẩy khớp nối lỏng lẻo bằng cách giữ cho các đối tượng không đề cập đến nhau một cách rõ ràng và nó cho phép bạn thay đổi tương tác của chúng một cách độc lập.

**Áp dụng:** Sử dụng Mediator pattern khi

* Một tập hợp các đối tượng giao tiếp theo những cách được xác định rõ ràng nhưng phức tạp. Kết quả là sự phụ thuộc lẫn nhau không có cấu trúc và khó hiểu.
* Việc sử dụng lại một đối tượng là khó khăn vì nó tham chiếu và giao tiếp với nhiều đối tượng khác.
* Một hành vi được phân phối giữa một số class nên có thể tùy chỉnh mà không cần nhiều phân class.

**Các thành phần chính:**

* Mediator (DialogDirector): Định nghĩa một giao diện để giao tiếp với các Colleague objects.
* ConcreteMediator (FontDialogDirector)

+ Thực hiện hành vi hợp tác bằng cách phối hợp các Colleague objects.

+ Biết và duy trì các Colleague của mình.

* Colleague classes (ListBox, EntryField)

+ Mỗi class Colleague biết đối tượng Mediator của nó.

+ Mỗi Colleague giao tiếp với Mediator của mình bất cứ khi nào nó có thể mặt khác giao tiếp với một Colleague khác.

**Ưu điểm:**

* Đảm bảo tính đa nhiệm. Bạn có thể trích xuất thông tin liên lạc giữa các thành phần khác nhau vào một nơi duy nhất, giúp việc hiểu và bảo trì dễ dàng hơn.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các mediators mới mà không phải thay đổi các thành phần thực tế.
* Bạn có thể giảm khớp nối giữa các thành phần khác nhau của chương trình.
* Bạn có thể tái sử dụng các thành phần riêng lẻ dễ dàng hơn.

**Nhược điểm:** Theo thời gian Mediator có thể tiến hóa thành God object.

**Cấu trúc:** A picture containing line, diagram, screenshot, font

Description automatically generatedMột cấu trúc đối tượng điển hình có thể trông như thế này: A diagram of a mediator

Description automatically generated with low confidence

**Template cho Python:**

"""

Define an object that encapsulates how a set of objects interact.

Mediator promotes loose coupling by keeping objects from referring to

each other explicitly, and it lets you vary their interaction

independently.

"""

class Mediator:

    """

    Implement cooperative behavior by coordinating Colleague objects.

    Know and maintains its colleagues.

    """

    def \_\_init\_\_(self):

        self.\_colleague\_1 = Colleague1(self)

        self.\_colleague\_2 = Colleague2(self)

class Colleague1:

    """

    Know its Mediator object.

    Communicate with its mediator whenever it would have otherwise

    communicated with another colleague.

    """

    def \_\_init\_\_(self, mediator):

        self.\_mediator = mediator

class Colleague2:

    """

    Know its Mediator object.

    Communicate with its mediator whenever it would have otherwise

    communicated with another colleague.

    """

    def \_\_init\_\_(self, mediator):

        self.\_mediator = mediator

def main():

    mediator = Mediator()

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

    main()

**Code ví dụ:**

class AirportMediator:

    """

    The Mediator class that coordinates communication between airplanes and the airport.

    """

    def \_\_init\_\_(self):

        self.airplanes = []

    def register\_airplane(self, airplane):

        self.airplanes.append(airplane)

    def request\_landing(self, airplane):

        if self.check\_runway\_availability():

            print(f"Landing permission granted for {airplane}.")

            self.reserve\_runway()

        else:

            print(f"Landing permission denied for {airplane}. Waiting for runway availability.")

    def runway\_available(self):

        print("Runway is now available.")

        self.reserve\_runway()

    def reserve\_runway(self):

        print("Reserving the runway for landing.")

    def check\_runway\_availability(self):

        # Simulate runway availability

        return len(self.airplanes) == 0

class Airplane:

    """

    The Colleague class representing an airplane.

    """

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

        self.name = name

        self.mediator = mediator

    def request\_landing(self):

        self.mediator.request\_landing(self)

    def runway\_available(self):

        print(f"{self.name}: Received notification that runway is available. Preparing for landing.")

def main():

    mediator = AirportMediator()

    airplane1 = Airplane("Airplane", mediator)

    airplane2 = Airplane("Helicopter", mediator)

    airplane3 = Airplane("Plane", mediator)

    mediator.register\_airplane(airplane1)

    mediator.register\_airplane(airplane2)

    mediator.register\_airplane(airplane3)

    airplane1.request\_landing()

    airplane2.request\_landing()

    airplane3.request\_landing()

    # Simulate runway becoming available

    mediator.runway\_available()

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

    main()

**Giải thích code:**

* Class AirportMediator được sử dụng như một Trung gian (Mediator). Nó theo dõi các máy bay đã đăng ký và xử lý giao tiếp giữa chúng và sân bay. Class này có các phương thức như register\_airplane để đăng ký máy bay, request\_landing để xử lý yêu cầu hạ cánh từ các máy bay, runway\_available để thông báo cho máy bay biết về tình trạng sẵn sàng của đường băng, reserve\_runway để đặt trước đường băng, và check\_runway\_availability để kiểm tra xem đường băng có sẵn cho việc hạ cánh không.
* Class Airplane đại diện cho một đồng nghiệp, tức là một máy bay. Mỗi máy bay được liên kết với trung gian trong quá trình khởi tạo. Class này có phương thức request\_landing cho phép máy bay yêu cầu hạ cánh bằng cách gọi phương thức tương ứng trong trung gian. Nó cũng có phương thức runway\_available được gọi bởi trung gian để thông báo cho máy bay khi đường băng sẵn sàng.
* Trong hàm main, chúng ta tạo một thể hiện của class AirportMediator được gọi là mediator. Chúng ta cũng tạo ba thể hiện của class Airplane được gọi là airplane1, airplane2 và airplane3. Sau đó, chúng ta đăng ký các máy bay này với trung gian bằng cách sử dụng phương thức register\_airplane.
* Sau khi đăng ký, mỗi máy bay yêu cầu hạ cánh bằng cách gọi phương thức request\_landing. Nếu đường băng có sẵn, trung gian cấp quyền hạ cánh bằng cách in một thông báo. Ngược lại, trung gian thông báo cho máy bay biết rằng quyền hạ cánh bị từ chối và máy bay cần đợi đường băng sẵn sàng.
* Cuối cùng, chúng ta mô phỏng việc đường băng trở thành sẵn sàng bằng cách gọi phương thức runway\_available trên trung gian. Điều này kích hoạt trung gian thông báo cho các máy bay bằng cách gọi phương thức runway\_available của chúng, nơi chúng in một thông báo cho biết rằng chúng đã nhận được thông báo và đang chuẩn bị hạ cánh.
* Khi bạn chạy mã, nó sẽ xuất ra giao tiếp giữa các máy bay và trung gian, thể hiện quyền hạ cánh và tình trạng sẵn sàng của đường băng cho mỗi máy bay.

### **Memento**

**Giới thiệu:**

* Memento Pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Memento Pattern là lưu trữ và khôi phục trạng thái của một đối tượng mà không tiết lộ chi tiết cấu trúc và triển khai của nó.
* Trong Memento Pattern, một đối tượng được gọi là "originator" (nguyên tố) chịu trách nhiệm cho việc tạo ra và khôi phục trạng thái. Đối tượng Memento đại diện cho một trạng thái cụ thể của Originator và chứa thông tin cần thiết để khôi phục lại trạng thái đó. Một đối tượng "caretaker" (người chăm sóc) được sử dụng để quản lý và lưu trữ các đối tượng Memento.
* Tuần suất sử dụng: Thấp.
* Memento Pattern còn được gọi là Token.

**Mục tiêu:** Không vi phạm tính đóng gói, nắm bắt và ngoại hóa trạng thái bên trong của đối tượng để đối tượng có thể được khôi phục về trạng thái này sau này.

**Áp dụng:** Sử dụng mẫu Memento khi

* Ảnh chụp nhanh (một phần) trạng thái của đối tượng phải được lưu để có thể khôi phục về trạng thái đó sau này và
* Giao diện trực tiếp để lấy trạng thái sẽ hiển thị chi tiết triển khai và phá vỡ sự đóng gói của đối tượng.

**Các thành phần chính:**

* Memento (SolverState)

+ Lưu trữ trạng thái bên trong của Originator object. Memento có thể lưu trữ nhiều hay ít trạng thái bên trong của Originator khi cần thiết theo quyết định Originator .

+ Bảo vệ chống truy cập bởi các đối tượng không phải là Originator. Mementos có hai giao diện hiệu quả. Caretaker nhìn thấy một giao diện hẹp đối với Memento - nó chỉ có thể truyền Memento cho các object khác. Ngược lại, Originator nhìn thấy một giao diện rộng, một giao diện cho phép nó truy cập tất cả dữ liệu cần thiết để tự khôi phục về trạng thái trước đó. Lý tưởng nhất là chỉ Originator đã tạo ra Memento mới được phép truy cập vào trạng thái bên trong của Memento.

* Originator (ConstraintSolver):

+ Tạo một Memento chứa ảnh chụp nhanh về trạng thái bên trong hiện tại của nó.

+ Sử dụng Memento để khôi phục trạng thái bên trong của nó.

* Caretaker (undo mechanism):

+ Chịu trách nhiệm bảo quản an toàn Memento.

+ Không bao giờ thao tác hoặc kiểm tra nội dung của Memento.

**Ưu điểm:**

* Bạn có thể tạo ảnh chụp nhanh về trạng thái của object mà không vi phạm tính đóng gói của nó.
* Bạn có thể đơn giản hóa code của Originator bằng cách cho phép Caretaker duy trì lịch sử trạng thái của Originator.

**Nhược điểm:**

* Ứng dụng có thể tiêu tốn nhiều RAM nếu Clients tạo Memento quá thường xuyên.
* Caretaker nên theo dõi vòng đời của Originator để có thể tiêu hủy những Memento lỗi thời.
* Hầu hết các ngôn ngữ lập trình động, chẳng hạn như PHP, Python và JavaScript, không thể đảm bảo rằng trạng thái trong Memento không bị ảnh hưởng.

**Cấu trúc:** A diagram of a state

Description automatically generated with low confidence

**Template cho Python:**

"""

Capture and externalize an object's internal state so that the object

can be restored to this state later, without violating encapsulation.

"""

import pickle

class Originator:

    """

    Create a memento containing a snapshot of its current internal

    state.

    Use the memento to restore its internal state.

    """

    def \_\_init\_\_(self):

        self.\_state = None

    def set\_memento(self, memento):

        previous\_state = pickle.loads(memento)

        vars(self).clear()

        vars(self).update(previous\_state)

    def create\_memento(self):

        return pickle.dumps(vars(self))

def main():

    originator = Originator()

    memento = originator.create\_memento()

    originator.\_state = True

    originator.set\_memento(memento)

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

    main()

**Code ví dụ:**

class Player:

    """

    The Player class represents the game player whose state needs to be saved and restored.

    """

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

        self.name = name

        self.level = level

    def set\_level(self, level):

        print(f"Setting {self.name}'s level to:", level)

        self.level = level

    def save\_state(self):

        print(f"Saving {self.name}'s state.")

        return Memento(self.level)

    def restore\_state(self, memento):

        self.level = memento.get\_state()

        print(f"Restoring {self.name}'s level to:", self.level)

class Memento:

    """

    The Memento class stores the state of the Player.

    """

    def \_\_init\_\_(self, state):

        self.state = state

    def get\_state(self):

        return self.state

class Caretaker:

    """

    The Caretaker class manages the mementos of the Player.

    """

    def \_\_init\_\_(self):

        self.mementos = []

    def add\_memento(self, memento):

        self.mementos.append(memento)

    def get\_memento(self, index):

        return self.mementos[index]

def main():

    player = Player("John", 1)

    caretaker = Caretaker()

    # Set initial level and save state

    player.set\_level(2)

    caretaker.add\_memento(player.save\_state())

    # Increase level and save state

    player.set\_level(3)

    caretaker.add\_memento(player.save\_state())

    # Decrease level

    player.set\_level(2)

    # Restore to previous state

    memento = caretaker.get\_memento(1)

    player.restore\_state(memento)

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

    main()

**Giải thích code:**

* Class Player đại diện cho người chơi trong trò chơi. Nó có các thuộc tính name và level để lưu trữ tên người chơi và cấp độ hiện tại. Phương thức set\_level được sử dụng để cập nhật cấp độ của người chơi, và phương thức save\_state tạo ra một đối tượng Memento với cấp độ hiện tại và trả về nó. Phương thức restore\_state nhận một đối tượng Memento và khôi phục cấp độ của người chơi về trạng thái đã lưu.
* Class Memento có nhiệm vụ lưu trữ trạng thái của người chơi. Nó có một constructor nhận cấp độ của người chơi làm đối số và một phương thức get\_state để truy xuất cấp độ đã lưu.
* Class Caretaker quản lý tập hợp các memento. Nó có một danh sách gọi là mementos để lưu trữ các memento được tạo bởi người chơi. Phương thức add\_memento thêm một memento vào danh sách, và phương thức get\_memento lấy một memento từ danh sách dựa trên một chỉ mục.
* Trong hàm main, chúng ta tạo một đối tượng Player có tên là "John" với cấp độ ban đầu là 1, và một đối tượng Caretaker để quản lý các memento. Sau đó, chúng ta đặt cấp độ của người chơi thành 2 và lưu trạng thái như một memento. Sau khi tăng cấp độ lên 3 và lưu trạng thái một lần nữa, chúng ta giảm cấp độ xuống 2. Cuối cùng, chúng ta lấy memento thứ hai từ caretaker và khôi phục cấp độ của người chơi về trạng thái đã lưu.
* Khi chạy code, nó sẽ xuất ra các hành động được thực hiện, bao gồm thiết lập cấp độ, lưu trạng thái và khôi phục trạng thái. Kết quả sẽ thể hiện cách cấp độ của người chơi được lưu trữ và khôi phục bằng cách sử dụng mẫu thiết kế Memento.

### **Observer**

**Giới thiệu:**

* Observer pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Observer pattern là thiết lập một mối quan hệ phụ thuộc một-nhiều giữa các đối tượng, sao cho khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc sẽ được thông báo và cập nhật trạng thái tương ứng.
* Tuần suất sử dụng: Cao.
* Observer Pattern còn được gọi là Dependents, Publish-Subscribe.

**Mục tiêu:** Xác định sự phụ thuộc một-nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái, tất cả các đối tượng phụ thuộc của nó sẽ được thông báo và cập nhật tự động.

**Áp dụng:** Sử dụng Observer Pattern trong bất kỳ trường hợp nào sau đây:

* Khi một trừu tượng có hai khía cạnh, một khía cạnh phụ thuộc vào khía cạnh kia. Đóng gói các khía cạnh này trong các đối tượng riêng biệt cho phép bạn thay đổi và sử dụng lại chúng một cách độc lập.
* Khi thay đổi một đối tượng đòi hỏi phải thay đổi những đối tượng khác mà bạn không biết bao nhiêu đối tượng cần phải được thay đổi.
* Khi một đối tượng có thể thông báo cho các đối tượng khác mà không cần đưa ra giả định về những đối tượng này là ai. Nói cách khác, bạn không muốn những các đối tượng liên kết chặt chẽ với nhau.

**Các thành phần chính:**

* Subject:

+ Biết người quan sát của nó. Bất kỳ số lượng Observer objects nào cũng có thể quan sát một chủ đề.

+ Cung cấp giao diện để gắn và tách các Observer objects.

* Observer: xác định giao diện cập nhật cho các đối tượng sẽ được thông báo về các thay đổi trong một subject.
* ConcreteSubject

+ Lưu trữ trạng thái quan tâm đến các đối tượng ConcreteObserver.

+ Gửi thông báo cho Observer khi trạng thái của nó thay đổi.

* ConcreteObserver:

+ Duy trì một tham chiếu đến một đối tượng ConcreteSubject.

+ Trạng thái cửa hàng nên phù hợp với subject.

+ Triển khai giao diện cập nhật của Observer để giữ trạng thái của nó nhất quán với subject.

**Ưu điểm:**

* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các Observer classes mới mà không phải thay đổi mã của Subject (và ngược lại nếu có giao diện của subject).
* Bạn có thể thiết lập quan hệ giữa các đối tượng trong thời gian chạy.

**Nhược điểm:** Observer được thông báo theo thứ tự ngẫu nhiên.

**Cấu trúc:** A picture containing text, diagram, plan, font

Description automatically generated

**Template cho Python:**

"""

Define a one-to-many dependency between objects so that when one object

changes state, all its dependents are notified and updatedautomatically.

"""

import abc

class Subject:

    """

    Know its observers. Any number of Observer objects may observe a

    subject.

    Send a notification to its observers when its state changes.

    """

    def \_\_init\_\_(self):

        self.\_observers = set()

        self.\_subject\_state = None

    def attach(self, observer):

        observer.\_subject = self

        self.\_observers.add(observer)

    def detach(self, observer):

        observer.\_subject = None

        self.\_observers.discard(observer)

    def \_notify(self):

        for observer in self.\_observers:

            observer.update(self.\_subject\_state)

    @property

    def subject\_state(self):

        return self.\_subject\_state

    @subject\_state.setter

    def subject\_state(self, arg):

        self.\_subject\_state = arg

        self.\_notify()

class Observer(metaclass=abc.ABCMeta):

    """

    Define an updating interface for objects that should be notified of

    changes in a subject.

    """

    def \_\_init\_\_(self):

        self.\_subject = None

        self.\_observer\_state = None

    @abc.abstractmethod

    def update(self, arg):

        pass

class ConcreteObserver(Observer):

    """

    Implement the Observer updating interface to keep its state

    consistent with the subject's.

    Store state that should stay consistent with the subject's.

    """

    def update(self, arg):

        self.\_observer\_state = arg

        # ...

def main():

    subject = Subject()

    concrete\_observer = ConcreteObserver()

    subject.attach(concrete\_observer)

    subject.subject\_state = 123

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

    main()

**Code ví dụ:**

import abc

class YouTubeChannel:

    """

    The Subject class representing a YouTube channel.

    """

    def \_\_init\_\_(self, channel\_name):

        self.channel\_name = channel\_name

        self.subscribers = []

    def subscribe(self, subscriber):

        self.subscribers.append(subscriber)

    def unsubscribe(self, subscriber):

        self.subscribers.remove(subscriber)

    def notify\_subscribers(self, video\_title):

        for subscriber in self.subscribers:

            subscriber.update(self.channel\_name, video\_title)

class Subscriber:

    """

    The Observer class representing a subscriber to a YouTube channel.

    """

    def \_\_init\_\_(self, name):

        self.name = name

    def update(self, channel\_name, video\_title):

        print(f"{self.name} received a new video '{video\_title}' from {channel\_name}.")

def main():

    channel = YouTubeChannel("MyAwesomeChannel")

    subscriber1 = Subscriber("John")

    subscriber2 = Subscriber("Alice")

    channel.subscribe(subscriber1)

    channel.subscribe(subscriber2)

    channel.notify\_subscribers("New Video: Introduction to Python")

    channel.unsubscribe(subscriber2)

    channel.notify\_subscribers("New Video: Advanced Machine Learning")

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

    main()

**Giải thích code:**

* Class YouTubeChannel là class Subject, đại diện cho một kênh YouTube. Nó có một thuộc tính channel\_name và lưu trữ danh sách các người đăng ký. Phương thức subscribe cho phép người đăng ký đăng ký vào kênh bằng cách thêm họ vào danh sách. Phương thức unsubscribe loại bỏ một người đăng ký khỏi danh sách. Phương thức notify\_subscribers có nhiệm vụ thông báo cho tất cả người đăng ký bằng cách gọi phương thức update của họ và truyền tên kênh và tiêu đề video mới.
* Class Subscriber là class Observer, đại diện cho người đăng ký kênh YouTube. Mỗi người đăng ký có một tên và định nghĩa phương thức update, được gọi khi một video mới được xuất bản trên kênh mà họ đang đăng ký. Trong ví dụ này, phương thức update đơn giản là in ra tên của người đăng ký, cùng với tên kênh và tiêu đề video mới.
* Trong hàm main, chúng ta tạo một thể hiện của class YouTubeChannel được gọi là channel. Chúng ta cũng tạo hai thể hiện của class Subscriber là subscriber1 và subscriber2. Tiếp theo, chúng ta thêm cả hai người đăng ký vào kênh bằng cách sử dụng phương thức subscribe.
* Sau đó, chúng ta gọi phương thức notify\_subscribers trên kênh, truyền tiêu đề của video mới là đối số. Điều này kích hoạt kênh thông báo cho tất cả người đăng ký, và phương thức update của mỗi người đăng ký được gọi, in ra thông tin video nhận được cùng với tên người đăng ký.
* Tiếp theo, chúng ta hủy đăng ký subscriber2 khỏi kênh bằng cách sử dụng phương thức unsubscribe. Cuối cùng, chúng ta gọi lại phương thức notify\_subscribers để xuất bản một video khác. Lần này, chỉ subscriber1 sẽ nhận được thông báo vì subscriber2 đã bị hủy đăng ký.
* Khi bạn chạy code, nó sẽ xuất ra thông báo về video mới nhận được bởi các người đăng ký, thể hiện mô hình Observer trong việc đăng ký một kênh YouTube.

### **State**

**Giới thiệu:**

* State pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của State pattern là cho phép một đối tượng thay đổi hành vi của mình khi trạng thái nội tại của nó thay đổi, mà không làm thay đổi cấu trúc của đối tượng.
* Trong State pattern, mỗi trạng thái của đối tượng được đại diện bởi một đối tượng con gọi là "state" (trạng thái). Các trạng thái này thực hiện cùng một giao diện (interface) và chứa các phương thức để xử lý hành vi tương ứng với trạng thái đó. Đối tượng chứa trạng thái được gọi là "context" (bối cảnh) và chuyển đổi giữa các trạng thái bằng cách thay đổi đối tượng state hiện tại.
* Tuần suất sử dụng: Trung bình.
* State Pattern còn được gọi là Objects for States.

**Mục tiêu:** Cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi class của nó.

**Áp dụng:** Sử dụng mẫu State trong một trong các trường hợp sau:

* Hành vi của một đối tượng phụ thuộc vào trạng thái của nó và nó phải thay đổi hành vi của mình trong thời gian chạy tùy thuộc vào trạng thái đó.
* Các phép toán có các câu lệnh điều kiện lớn, nhiều phần phụ thuộc vào trạng thái của đối tượng. Trạng thái này thường được biểu diễn bằng một hoặc nhiều hằng số liệt kê. Thông thường, một số thao tác sẽ chứa cấu trúc điều kiện giống nhau này. State Pattern đặt mỗi nhánh của điều kiện trong một class riêng biệt. Điều này cho phép bạn coi trạng thái của đối tượng là một đối tượng theo đúng nghĩa của nó, có thể thay đổi độc lập với các đối tượng khác

**Các thành phần chính:**

* Context (TCPConnection)

+ Xác định giao diện quan tâm đến Clients.

+ Duy trì một thể hiện của class con ConcreteState xác định trạng thái hiện tại.

* State (TCPState)

+ Xác định một giao diện để đóng gói hành vi được liên kết với một trạng thái cụ thể của Context.

* ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed): mỗi class con thực hiện hành vi được liên kết với một trạng thái của Context.

**Ưu điểm:**

* Đảm bảo tính đa hình. Tổ chức mã liên quan đến các trạng thái cụ thể thành các classes riêng biệt.
* Đảm bảo nguyên tắc Mở/Đóng. Giới thiệu các trạng thái mới mà không thay đổi các class trạng thái hiện có hoặc Context.
* Đơn giản hóa mã ngữ cảnh bằng cách loại bỏ các điều kiện máy trạng thái cồng kềnh.

**Nhược điểm:** Việc áp dụng mẫu có thể là quá mức cần thiết nếu máy trạng thái chỉ có một vài trạng thái hoặc hiếm khi thay đổi.

**Cấu trúc:** A diagram of a concrete state

Description automatically generated with medium confidence

**Template cho Python:**

"""

Allow an object to alter its behavior when its internal state changes.

The object will appear to change its class.

"""

import abc

class Context:

    """

    Define the interface of interest to clients.

    Maintain an instance of a ConcreteState subclass that defines the

    current state.

    """

    def \_\_init\_\_(self, state):

        self.\_state = state

    def request(self):

        self.\_state.handle()

class State(metaclass=abc.ABCMeta):

    """

    Define an interface for encapsulating the behavior associated with a

    particular state of the Context.

    """

    @abc.abstractmethod

    def handle(self):

        pass

class ConcreteStateA(State):

    """

    Implement a behavior associated with a state of the Context.

    """

    def handle(self):

        pass

class ConcreteStateB(State):

    """

    Implement a behavior associated with a state of the Context.

    """

    def handle(self):

        pass

def main():

    concrete\_state\_a = ConcreteStateA()

    context = Context(concrete\_state\_a)

    context.request()

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

    main()

**Code ví dụ:**

import abc

class PhoneState(metaclass=abc.ABCMeta):

    """

    The State interface that defines the contract for different phone states.

    """

    @abc.abstractmethod

    def press\_button(self):

        pass

class UnlockedState(PhoneState):

    """

    A concrete state representing the unlocked state of the phone.

    """

    def press\_button(self):

        print("Executing various functions.")

class LockedState(PhoneState):

    """

    A concrete state representing the locked state of the phone.

    """

    def press\_button(self):

        print("Showing unlock screen.")

class LowBatteryState(PhoneState):

    """

    A concrete state representing the low battery state of the phone.

    """

    def press\_button(self):

        print("Showing charging screen.")

class Phone:

    """

    The Context class that holds a reference to the current state of the phone.

    """

    def \_\_init\_\_(self):

        self.current\_state = UnlockedState()

    def set\_state(self, state):

        self.current\_state = state

    def press\_button(self):

        self.current\_state.press\_button()

def main():

    phone = Phone()

    # Test with unlocked state

    phone.press\_button()

    # Change to locked state

    phone.set\_state(LockedState())

    phone.press\_button()

    # Change to low battery state

    phone.set\_state(LowBatteryState())

    phone.press\_button()

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

    main()

**Giải thích code:**

* PhoneState là một class trừu tượng đại diện cho giao diện cho các trạng thái khác nhau của điện thoại. Nó có một phương thức press\_button() sẽ được triển khai bởi các class trạng thái cụ thể.
* UnlockedState, LockedState, và LowBatteryState là các class trạng thái cụ thể đại diện cho trạng thái mở khóa, trạng thái khóa và trạng thái pin yếu của điện thoại, tương ứng. Mỗi class này cung cấp triển khai riêng của phương thức press\_button() dựa trên hành vi cụ thể liên quan đến trạng thái đó.
* Phone là class ngữ cảnh (context) giữ một tham chiếu đến trạng thái hiện tại của điện thoại. Nó có một phương thức set\_state() để thay đổi trạng thái hiện tại và một phương thức press\_button() để chuyển tiếp hành động nhấn nút cho đối tượng trạng thái hiện tại.
* Trong hàm main(), chúng ta tạo một thể hiện của class Phone gọi là phone. Ban đầu, điện thoại ở trạng thái mở khóa, vì vậy khi chúng ta gọi phone.press\_button(), nó thực thi các chức năng khác nhau. Sau đó, chúng ta thay đổi trạng thái thành khóa và gọi phone.press\_button(), điều này sẽ hiển thị màn hình mở khóa. Cuối cùng, chúng ta thay đổi trạng thái thành pin yếu và gọi phone.press\_button(), nó sẽ hiển thị màn hình sạc pin.

### **Strategy**

**Giới thiệu:**

* Strategy pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Strategy pattern là cho phép đối tượng chứa thay đổi thuật toán được sử dụng trong runtime một cách linh hoạt, mà không phải thay đổi cấu trúc của đối tượng.
* Trong Strategy pattern, các thuật toán được đóng gói trong các đối tượng riêng biệt được gọi là "strategy" (chiến lược). Mỗi strategy triển khai một giao diện chung, giúp đối tượng chứa có thể thay đổi strategy mà không phụ thuộc vào cụ thể của từng strategy. Đối tượng chứa được gọi là "context" (bối cảnh) và chứa một tham chiếu đến một strategy cụ thể. Khi cần thay đổi thuật toán, context có thể thay đổi strategy bằng cách thay đổi tham chiếu của mình.
* Tuần suất sử dụng: Cao Trung bình.
* Strategy Pattern còn được gọi là Policy.

**Mục tiêu:** Định nghĩa một nhóm các thuật toán, đóng gói từng thuật toán và làm cho chúng có thể hoán đổi cho nhau. Strategy cho phép thuật toán thay đổi độc lập với các máy khách sử dụng nó.

**Áp dụng:** Sử dụng Strategy Pattern khi:

* Nhiều class liên quan chỉ khác nhau về hành vi của chúng. Strategy cung cấp một cách để định cấu hình một class với một trong nhiều hành vi.
* Bạn cần các biến thể khác nhau của thuật toán. Ví dụ: bạn có thể xác định các thuật toán phản ánh sự đánh đổi không gian/thời gian khác nhau. Strategy có thể được sử dụng khi các biến thể này được triển khai dưới dạng phân cấp class của các thuật toán.
* Thuật toán sử dụng dữ liệu mà Clients không nên biết. Sử dụng Strategy Pattern để tránh làm lộ cấu trúc dữ liệu phức tạp, dành riêng cho thuật toán.
* Một class định nghĩa nhiều hành vi và những hành vi này xuất hiện dưới dạng nhiều câu lệnh điều kiện trong các hoạt động của nó. Thay vì nhiều điều kiện, hãy di chuyển các nhánh điều kiện có liên quan vào Strategy class của riêng chúng.

**Các thành phần chính:**

* Strategy (Compositor): Tuyên bố một giao diện chung cho tất cả các thuật toán được hỗ trợ. Bối cảnh sử dụng giao diện này để gọi thuật toán được xác định bởi ConcreteStrategy.
* ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor): Thực hiện thuật toán bằng giao diện Strategy.
* Context (Composition):

+ Được cấu hình với một đối tượng ConcreteStrategy.

+ Duy trì một tham chiếu đến một đối tượng Strategy.

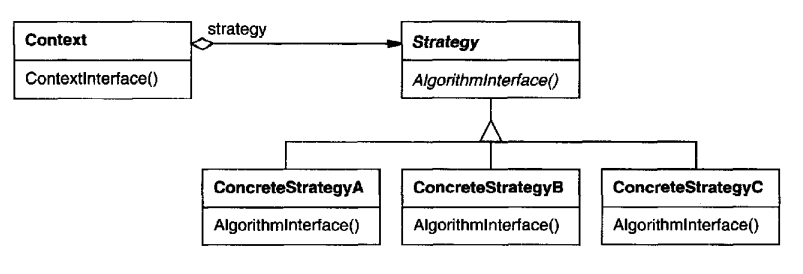
+ Có thể xác định giao diện cho phép Strategy truy cập dữ liệu của nó.

**Ưu điểm:**

* Bạn có thể trao đổi các thuật toán được sử dụng bên trong một đối tượng khi chạy.
* Bạn có thể tách biệt các chi tiết triển khai của một thuật toán khỏi mã sử dụng thuật toán đó.
* Bạn có thể thay thế thừa kế bằng thành phần.
* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các Strategy mới mà không phải thay đổi bối cảnh.

**Nhược điểm:**

* Nếu bạn chỉ có một vài thuật toán và chúng hiếm khi thay đổi, thì không có lý do thực sự nào để phức tạp hóa chương trình với các class và giao diện mới đi kèm với mẫu.
* Clients phải nhận thức được sự khác biệt giữa các Strategy để có thể chọn một Strategy phù hợp.
* Rất nhiều ngôn ngữ lập trình hiện đại có hỗ trợ kiểu chức năng cho phép bạn triển khai các phiên bản khác nhau của thuật toán bên trong một tập hợp các hàm ẩn danh. Sau đó, bạn có thể sử dụng các chức năng này chính xác như bạn đã sử dụng các Strategy objects, nhưng không làm code của bạn trở nên cồng kềnh với các class và giao diện bổ sung.

**Cấu trúc:** 

**Template cho Python:**

"""

Define a family of algorithms, encapsulate each one, and make them

interchangeable. Strategy lets the algorithm vary independently from

clients that use it.

"""

import abc

class Context:

    """

    Define the interface of interest to clients.

    Maintain a reference to a Strategy object.

    """

    def \_\_init\_\_(self, strategy):

        self.\_strategy = strategy

    def context\_interface(self):

        self.\_strategy.algorithm\_interface()

class Strategy(metaclass=abc.ABCMeta):

    """

    Declare an interface common to all supported algorithms. Context

    uses this interface to call the algorithm defined by a

    ConcreteStrategy.

    """

    @abc.abstractmethod

    def algorithm\_interface(self):

        pass

class ConcreteStrategyA(Strategy):

    """

    Implement the algorithm using the Strategy interface.

    """

    def algorithm\_interface(self):

        pass

class ConcreteStrategyB(Strategy):

    """

    Implement the algorithm using the Strategy interface.

    """

    def algorithm\_interface(self):

        pass

def main():

    concrete\_strategy\_a = ConcreteStrategyA()

    context = Context(concrete\_strategy\_a)

    context.context\_interface()

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

    main()

**Code ví dụ:**

class TourStrategy:

    """

    The Strategy interface that defines the contract for tour strategies.

    """

    def plan\_tour(self):

        pass

class WalkingTourStrategy(TourStrategy):

    """

    A concrete strategy that represents a walking tour.

    """

    def plan\_tour(self):

        print("Planning a walking tour around Ho Chi Minh City.")

        # Logic for planning a walking tour

class BikingTourStrategy(TourStrategy):

    """

    A concrete strategy that represents a biking tour.

    """

    def plan\_tour(self):

        print("Planning a biking tour around Ho Chi Minh City.")

        # Logic for planning a biking tour

class CarTourStrategy(TourStrategy):

    """

    A concrete strategy that represents a car tour.

    """

    def plan\_tour(self):

        print("Planning a car tour around Ho Chi Minh City.")

        # Logic for planning a car tour

class HoChiMinhCityTour:

    """

    The Context class that holds a reference to the current tour strategy.

    """

    def \_\_init\_\_(self, tour\_strategy):

        self.tour\_strategy = tour\_strategy

    def set\_tour\_strategy(self, tour\_strategy):

        self.tour\_strategy = tour\_strategy

    def plan\_tour(self):

        self.tour\_strategy.plan\_tour()

def main():

    # Create the tour strategies

    walking\_tour\_strategy = WalkingTourStrategy()

    biking\_tour\_strategy = BikingTourStrategy()

    car\_tour\_strategy = CarTourStrategy()

    # Create the Ho Chi Minh City tour with the default strategy

    hcm\_city\_tour = HoChiMinhCityTour(walking\_tour\_strategy)

    # Plan the tour using different strategies

    hcm\_city\_tour.plan\_tour()

    hcm\_city\_tour.set\_tour\_strategy(biking\_tour\_strategy)

    hcm\_city\_tour.plan\_tour()

    hcm\_city\_tour.set\_tour\_strategy(car\_tour\_strategy)

    hcm\_city\_tour.plan\_tour()

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

    main()

**Giải thích code:**

* Class TourStrategy là một giao diện (interface) định nghĩa hợp đồng cho các chiến lược đi tour. Nó chỉ có một phương thức plan\_tour() không có triển khai cụ thể.
* Các class WalkingTourStrategy, BikingTourStrategy, và CarTourStrategy là các class con của TourStrategy, triển khai phương thức plan\_tour() theo cách cụ thể cho từng loại tour.
* Class HoChiMinhCityTour là class Context (bối cảnh) và giữ một tham chiếu đến chiến lược đi tour hiện tại thông qua thuộc tính tour\_strategy. Nó cung cấp phương thức set\_tour\_strategy() để thay đổi chiến lược đi tour và phương thức plan\_tour() để kế hoạch và thực hiện tour theo chiến lược hiện tại.
* Trong hàm main(), chúng ta tạo các chiến lược đi tour khác nhau: walking\_tour\_strategy, biking\_tour\_strategy, và car\_tour\_strategy. Sau đó, chúng ta tạo một tour TP Hồ Chí Minh với chiến lược đi tour mặc định là walking\_tour\_strategy.
* Chúng ta gọi phương thức plan\_tour() trên hcm\_city\_tour để kế hoạch và thực hiện tour theo chiến lược hiện tại (trong trường hợp đầu tiên là đi bộ). Sau đó, chúng ta thay đổi chiến lược đi tour bằng cách gọi phương thức set\_tour\_strategy() và tiếp tục gọi plan\_tour() để thực hiện các tour khác nhau (trong trường hợp thứ hai là đi xe đạp, và trong trường hợp thứ ba là đi xe hơi).
* Kết quả của chương trình sẽ là in ra thông tin về việc kế hoạch tour theo từng chiến lược đi tour khác nhau xung quanh TP Hồ Chí Minh.

### **Template Method**

**Giới thiệu:**

* Template Method là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Template Method pattern là định nghĩa một khung sườn (template) cho một thuật toán và cho phép các bước cụ thể của thuật toán được triển khai bởi các class con.
* Trong Template Method pattern, một class cha chứa một phương thức gọi là "template method" (phương thức mẫu). Phương thức này xác định các bước cơ bản của thuật toán và gọi các phương thức con cụ thể để thực hiện từng bước. Các phương thức con được gọi là "hook methods" và chúng được triển khai bởi các class con để cung cấp các bước cụ thể của thuật toán.
* Tuần suất sử dụng: Trung bình.

**Mục tiêu:** Xác định khung của một thuật toán trong một hoạt động, trì hoãn một số bước cho các class con. Template Methdod Pattern cho phép các class con xác định lại các bước nhất định của thuật toán mà không thay đổi cấu trúc của thuật toán.

**Áp dụng:** Nên sử dụng mẫu Phương thức mẫu:

* Để thực hiện các phần bất biến của thuật toán một lần và để nó cho các class con thực hiện hành vi có thể thay đổi.
* Khi hành vi chung giữa các class con nên được phân tích và bản địa hóa trong một class chung để tránh trùng lặp code. Đây là một ví dụ điển hình về "tái cấu trúc để khái quát hóa" như được mô tả bởi Opdyke và Johnson. Trước tiên, bạn xác định sự khác biệt trong code hiện có và sau đó tách sự khác biệt thành các hoạt động mới. Cuối cùng, bạn thay thế mã khác bằng một Template Method gọi một trong các thao tác mới này.
* Để kiểm soát các phần mở rộng của class con. Bạn có thể xác định một phương thức mẫu gọi các thao tác "hook" (xem Hậu quả) tại các điểm cụ thể, do đó chỉ cho phép các phần mở rộng tại các điểm đó.

**Các thành phần chính:**

* AbstractClass (Application)

+ Định nghĩa các phép toán nguyên thủy trừu tượng mà các class con cụ thể định nghĩa để thực hiện các bước của thuật toán.

+ Thực hiện một phương pháp khuôn mẫu xác định bộ xương của một thuật toán. Template gọi các thao tác nguyên thủy cũng như các thao tác được định nghĩa trong AbstractClass hoặc của các objects khác.

* ConcreteClass (MyApplication): thực hiện các hoạt động nguyên thủy để thực hiện phân class cụ thể các bước của thuật toán.

**Ưu điểm:**

* Bạn có thể cho phép máy khách chỉ ghi đè một số phần nhất định của một thuật toán lớn, giúp chúng ít bị ảnh hưởng bởi những thay đổi xảy ra với các phần khác của thuật toán.
* Bạn có thể kéo mã trùng lặp vào một siêu class.

**Nhược điểm:**

* Một số clients có thể bị giới hạn bởi khung thuật toán được cung cấp.
* Bạn có thể vi phạm Nguyên tắc thay thế Liskov bằng cách ngăn chặn việc triển khai bước mặc định thông qua một class con.
* Các phương thức mẫu có xu hướng khó duy trì hơn khi chúng có nhiều bước hơn.

**Cấu trúc:** A diagram of a process

Description automatically generated with low confidence

**Template cho Python:**

"""

Define the skeleton of an algorithm in an operation, deferring some

steps to subclasses. Template Method lets subclasses redefine certain

steps of an algorithm without changing the algorithm's structure.

"""

import abc

class AbstractClass(metaclass=abc.ABCMeta):

    """

    Define abstract primitive operations that concrete subclasses define

    to implement steps of an algorithm.

    Implement a template method defining the skeleton of an algorithm.

    The template method calls primitive operations as well as operations

    defined in AbstractClass or those of other objects.

    """

    def template\_method(self):

        self.\_primitive\_operation\_1()

        self.\_primitive\_operation\_2()

    @abc.abstractmethod

    def \_primitive\_operation\_1(self):

        pass

    @abc.abstractmethod

    def \_primitive\_operation\_2(self):

        pass

class ConcreteClass(AbstractClass):

    """

    Implement the primitive operations to carry out

    subclass-specificsteps of the algorithm.

    """

    def \_primitive\_operation\_1(self):

        pass

    def \_primitive\_operation\_2(self):

        pass

def main():

    concrete\_class = ConcreteClass()

    concrete\_class.template\_method()

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

    main()

**Code ví dụ:**

import abc

class HouseConstructionTemplate(metaclass=abc.ABCMeta):

    """

    The abstract class that defines the template method for house construction.

    """

    def construct\_house(self):

        self.lay\_foundation()

        self.build\_frame()

        self.build\_walls()

        self.install\_plumbing()

        self.install\_electrical\_wiring()

        self.finish\_house()

    @abc.abstractmethod

    def lay\_foundation(self):

        pass

    @abc.abstractmethod

    def build\_frame(self):

        pass

    @abc.abstractmethod

    def build\_walls(self):

        pass

    @abc.abstractmethod

    def install\_plumbing(self):

        pass

    @abc.abstractmethod

    def install\_electrical\_wiring(self):

        pass

    def finish\_house(self):

        print("Finishing the house.")

class StandardHouseConstruction(HouseConstructionTemplate):

    """

    A concrete class representing the construction of a standard house.

    """

    def lay\_foundation(self):

        print("Laying the foundation for a standard house.")

    def build\_frame(self):

        print("Building the frame of a standard house.")

    def build\_walls(self):

        print("Building the walls of a standard house.")

    def install\_plumbing(self):

        print("Installing plumbing for a standard house.")

    def install\_electrical\_wiring(self):

        print("Installing electrical wiring for a standard house.")

class CustomHouseConstruction(HouseConstructionTemplate):

    """

    A concrete class representing the construction of a custom house.

    """

    def lay\_foundation(self):

        print("Laying the foundation for a custom house.")

    def build\_frame(self):

        print("Building the frame of a custom house with specific modifications.")

    def build\_walls(self):

        print("Building the walls of a custom house with specific modifications.")

    def install\_plumbing(self):

        print("Installing plumbing for a custom house with specific requirements.")

    def install\_electrical\_wiring(self):

        print("Installing electrical wiring for a custom house with specific requirements.")

def main():

    standard\_house = StandardHouseConstruction()

    standard\_house.construct\_house()

    print("---")

    custom\_house = CustomHouseConstruction()

    custom\_house.construct\_house()

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

    main()

**Giải thích code:**

* HouseConstructionTemplate là một class trừu tượng (abstract class) định nghĩa phương thức mẫu (construct\_house()) cho quy trình xây dựng nhà. Phương thức này đại diện cho toàn bộ quy trình xây dựng và gọi nhiều phương thức trừu tượng và phi trừu tượng cho từng bước xây dựng. Các phương thức trừu tượng đại diện cho các điểm mở rộng (extension points) có thể được tùy chỉnh bởi các class con.
* StandardHouseConstruction và CustomHouseConstruction là các class cụ thể (concrete classes) kế thừa từ HouseConstructionTemplate. Chúng cung cấp các triển khai cụ thể cho các phương thức trừu tượng, cho phép tùy chỉnh các bước xây dựng cho nhà tiêu chuẩn và nhà tùy chỉnh, tương ứng.
* Hàm main() thể hiện việc sử dụng mẫu Template Method. Nó tạo ra các thể hiện của StandardHouseConstruction và CustomHouseConstruction và gọi phương thức construct\_house() trên mỗi đối tượng. Kết quả đầu ra hiển thị chuỗi các bước xây dựng được thực hiện theo phương pháp mẫu.

### **Visitor**

**Giới thiệu:**

* Visitor pattern là một mẫu thiết kế (design pattern) thuộc nhóm Behavioral Patterns trong lĩnh vực phát triển phần mềm. Mục tiêu chính của Visitor pattern là tách rời việc xử lý các thao tác trên các đối tượng từ cấu trúc của chúng, đồng thời cho phép thêm các thao tác mới mà không cần thay đổi cấu trúc của các đối tượng đang xử lý.
* Trong Visitor pattern, chúng ta có hai thành phần chính: Visitor (Khách thăm) và Element (Phần tử). Mỗi đối tượng Element có thể được "ghé thăm" bởi một đối tượng Visitor, và đối tượng Visitor sẽ thực hiện các thao tác (hoặc việc xử lý) trên Element đó.
* Tuần suất sử dụng: Trung bình.

**Mục tiêu:** Đại diện cho một thao tác được thực hiện trên các phần tử của cấu trúc đối tượng. Visitor cho phép bạn xác định một thao tác mới mà không thay đổi các class của các phần tử mà thao tác đó hoạt động.

**Áp dụng:** Sử dụng Visitorpattern khi:

* Một cấu trúc đối tượng chứa nhiều class đối tượng với các giao diện khác nhau, và bạn muốn thực hiện các thao tác trên các đối tượng này phụ thuộc vào các class cụ thể của chúng.
* Nhiều thao tác riêng biệt và không liên quan cần được thực hiện trên các đối tượng trong một cấu trúc đối tượng, và bạn muốn tránh "làm ô nhiễm" các class của chúng bằng các thao tác này. Visitor cho phép bạn giữ các hoạt động liên quan với nhau bằng cách định nghĩa chúng trong một class. Khi cấu trúc đối tượng được chia sẻ bởi nhiều ứng dụng, hãy sử dụng Visitor để đặt các thao tác chỉ trong những ứng dụng cần chúng.
* Các class định nghĩa cấu trúc đối tượng hiếm khi thay đổi, nhưng bạn thường muốn định nghĩa các thao tác mới trên cấu trúc. Việc thay đổi các class cấu trúc đối tượng yêu cầu xác định lại giao diện cho tất cả khách truy cập, điều này có khả năng gây tốn kém. Nếu các class cấu trúc đối tượng thay đổi thường xuyên, thì có lẽ tốt hơn là định nghĩa các thao tác trong các class đó.

**Các thành phần chính:**

* Visitor (NodeVisitor): Khai báo một thao tác Visit cho từng class ConcreteElement trong cấu trúc đối tượng. Tên và chữ ký của thao tác xác định class gửi yêu cầu Visit cho Visitor. Điều đó cho phép Visitor xác định class cụ thể của phần tử đang được truy cập. Sau đó, Visitor có thể truy cập phần tử trực tiếp thông qua giao diện cụ thể của nó.
* ConcreteVisitor (TypeCheckingVisitor): Thực hiện từng thao tác được khai báo bởi Visitor. Mỗi thao tác thực hiện một đoạn thuật toán được xác định cho class đối tượng tương ứng trong cấu trúc. ConcreteVisitor cung cấp ngữ cảnh cho thuật toán và lưu trữ trạng thái cục bộ của nó. Trạng thái này thường tích lũy kết quả trong quá trình di chuyển của cấu trúc.
* Element (Node): Xác định thao tác Accept lấy Visitor làm đối số.
* ConcreteElement (AssignmentNode,VariableRefNode): Thực hiện thao tác Accept lấy Visitor làm đối số.
* ObjectStructure (Program):

+ Có thể liệt kê các phần tử của nó.

+ Có thể cung cấp giao diện cấp cao để cho phép Visitor truy cập vào các phần tử của nó.

+ Có thể là Composite (Composite Pattern)hoặc tập hợp chẳng hạn như danh sách hoặc tập hợp..

**Ưu điểm:**

* Đảm bảo nguyên tắc Mở/Đóng. Bạn có thể giới thiệu một hành vi mới có thể làm việc với các đối tượng của các class khác nhau mà không làm thay đổi các class này.
* Đảm bảo nguyên tắc trách nhiệm duy nhất. Bạn có thể di chuyển nhiều phiên bản của cùng một hành vi vào cùng một class.
* Một đối tượng Visitor có thể tích lũy một số thông tin hữu ích khi làm việc với các đối tượng khác nhau. Điều này có thể hữu ích khi bạn muốn duyệt qua một số cấu trúc đối tượng phức tạp, chẳng hạn như object trê và áp dụng Visitor vào từng đối tượng của cấu trúc này.

**Nhược điểm:**

* Bạn cần cập nhật tất cả Visitor mỗi khi một class được thêm vào hoặc xóa khỏi hệ thống phân cấp phần tử.
* Visitor có thể thiếu quyền truy cập cần thiết vào các trường riêng tư và phương thức của các phần tử mà họ phải làm việc cùng.

**Cấu trúc:** A picture containing text, diagram, parallel, font

Description automatically generated

**Template cho Python:**

"""

Represent an operation to be performed on the elements of an object

structure. Visitor lets you define a new operation without changing the

classes of the elements on which it operates.

"""

import abc

class Element(metaclass=abc.ABCMeta):

    """

    Define an Accept operation that takes a visitor as an argument.

    """

    @abc.abstractmethod

    def accept(self, visitor):

        pass

class ConcreteElementA(Element):

    """

    Implement an Accept operation that takes a visitor as an argument.

    """

    def accept(self, visitor):

        visitor.visit\_concrete\_element\_a(self)

class ConcreteElementB(Element):

    """

    Implement an Accept operation that takes a visitor as an argument.

    """

    def accept(self, visitor):

        visitor.visit\_concrete\_element\_b(self)

class Visitor(metaclass=abc.ABCMeta):

    """

    Declare a Visit operation for each class of ConcreteElement in the

    object structure. The operation's name and signature identifies the

    class that sends the Visit request to the visitor. That lets the

    visitor determine the concrete class of the element being visited.

    Then the visitor can access the element directly through its

    particular interface.

    """

    @abc.abstractmethod

    def visit\_concrete\_element\_a(self, concrete\_element\_a):

        pass

    @abc.abstractmethod

    def visit\_concrete\_element\_b(self, concrete\_element\_b):

        pass

class ConcreteVisitor1(Visitor):

    """

    Implement each operation declared by Visitor. Each operation

    implements a fragment of the algorithm defined for the corresponding

    class of object in the structure. ConcreteVisitor provides the

    context for the algorithm and stores its local state. This state

    often accumulates results during the traversal of the structure.

    """

    def visit\_concrete\_element\_a(self, concrete\_element\_a):

        pass

    def visit\_concrete\_element\_b(self, concrete\_element\_b):

        pass

class ConcreteVisitor2(Visitor):

    """

    Implement each operation declared by Visitor. Each operation

    implements a fragment of the algorithm defined for the corresponding

    class of object in the structure. ConcreteVisitor provides the

    context for the algorithm and stores its local state. This state

    often accumulates results during the traversal of the structure.

    """

    def visit\_concrete\_element\_a(self, concrete\_element\_a):

        pass

    def visit\_concrete\_element\_b(self, concrete\_element\_b):

        pass

def main():

    concrete\_visitor\_1 = ConcreteVisitor1()

    concrete\_element\_a = ConcreteElementA()

    concrete\_element\_a.accept(concrete\_visitor\_1)

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

    main()

**Code ví dụ:**

import abc

class Building(metaclass=abc.ABCMeta):

    """

    The abstract base class that defines the accept method for accepting a visitor.

    """

    @abc.abstractmethod

    def accept(self, visitor):

        pass

class ResidentialBuilding(Building):

    """

    A concrete class representing a residential building.

    """

    def accept(self, visitor):

        visitor.visit\_residential\_building(self)

class Bank(Building):

    """

    A concrete class representing a bank building.

    """

    def accept(self, visitor):

        visitor.visit\_bank(self)

class CoffeeShop(Building):

    """

    A concrete class representing a coffee shop building.

    """

    def accept(self, visitor):

        visitor.visit\_coffee\_shop(self)

class InsuranceAgent:

    """

    The visitor class representing an insurance agent.

    """

    def visit\_residential\_building(self, building):

        print("Selling medical insurance to the residents.")

    def visit\_bank(self, building):

        print("Selling theft insurance to the bank.")

    def visit\_coffee\_shop(self, building):

        print("Selling fire and flood insurance to the coffee shop.")

def main():

    residential\_building = ResidentialBuilding()

    bank = Bank()

    coffee\_shop = CoffeeShop()

    insurance\_agent = InsuranceAgent()

    residential\_building.accept(insurance\_agent)

    bank.accept(insurance\_agent)

    coffee\_shop.accept(insurance\_agent)

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

    main()

**Giải thích code:**

* Class Building là một class cơ sở trừu tượng, định nghĩa phương thức accept để chấp nhận một visitor. Mỗi class xây dựng cụ thể (ResidentialBuilding, Bank, CoffeeShop) triển khai phương thức accept để cho phép một visitor ghé thăm tòa nhà.
* Class InsuranceAgent đại diện cho visitor, trong trường hợp này là một nhân viên bảo hiểm. Nó cung cấp các phương thức visit tương ứng cho mỗi loại tòa nhà. Khi một nhân viên bảo hiểm ghé thăm một tòa nhà, phương thức visit tương ứng được gọi.
* Trong hàm main, chúng ta tạo các đối tượng của các loại tòa nhà khác nhau (ResidentialBuilding, Bank, CoffeeShop) và một đối tượng của class InsuranceAgent. Sau đó, chúng ta gọi phương thức accept trên mỗi tòa nhà, truyền nhân viên bảo hiểm làm visitor. Điều này kích hoạt phương thức visit tương ứng trong nhân viên bảo hiểm và nhân viên bảo hiểm bán các chính sách bảo hiểm tương ứng dựa trên loại tòa nhà.

Hết

# **REFERENCE**

1. <https://gpcoder.com/4164-gioi-thieu-design-patterns/>
2. <https://sourcemaking.com/design_patterns>
3. <https://refactoring.guru/design-patterns>
4. Design Pattern, Tác giả: [Erich Gamma](https://www.google.com/search?client=opera&hs=Mlq&sa=X&sxsrf=APwXEdfhyfg8b6PB5bVXCq1YN006oA9gxA:1686754391408&q=Erich+Gamma&si=AMnBZoEofOODruSEFWFjdccePwMH96ZlZt3bOiKSR9t4pqlu2MldmXnjdSP7_QWjTt-wHCm5oWCWfAcSz8-tiCG5NDu99q4IXD72Dx-eVMj-qvSkURxPPEk6no1gWl4vfw5g1ARPY9YWXqFHQ_yptbjEackYnQedaH1QzehdPOwOLVY7ymcLYuPb5KaAHsg9pUdv1FIh7okOpth7pDfz9QImKm7OJWSh5O0RVjWvKp3tahYJJoeQIxY%3D&ved=2ahUKEwjjxsbigcP_AhXbnFYBHTquDeYQmxMoAHoECCQQAg), [John Vlissides](https://www.google.com/search?client=opera&hs=Mlq&sa=X&sxsrf=APwXEdfhyfg8b6PB5bVXCq1YN006oA9gxA:1686754391408&q=design+patterns+john+vlissides&si=AMnBZoEofOODruSEFWFjdccePwMH96ZlZt3bOiKSR9t4pqlu2ProOMYKkfAsHB-_5cstHuTiMP565sVymub4eUmOvzfUzmwemqN_T-XMWxpyhtwySclhBF-qsYNY56jwWnce73Izu3ehLIL6VVqcOEMlNcokctaP9dtmgZOb9jRRs2EpNFCej4O8itI3aDR9cLWibGpwrV09RilreSic2v20kSKiy__nvFlzhV_aPmsjb8GcPnEmFLjlQOLUthtXAInNUiDCswecHJMliSxMg3_0goWPCdUAkA%3D%3D&ved=2ahUKEwjjxsbigcP_AhXbnFYBHTquDeYQmxMoAXoECCQQAw), [Richard Helm](https://www.google.com/search?client=opera&hs=Mlq&sa=X&sxsrf=APwXEdfhyfg8b6PB5bVXCq1YN006oA9gxA:1686754391408&q=richard+helm&si=AMnBZoEofOODruSEFWFjdccePwMH96ZlZt3bOiKSR9t4pqlu2FW3xtrrfkk96tc8e_sTn0iTHuqu8bWwUTYn_tRTeRLP09SqunYC-7JHmEWR1aCCJQyxO4OSYt1HSmnjKnjZcYO7IPGPsKtMkeUIS7d9hWf-WTkZeMqJ5IpscWIh1KBbNw26F0_6Lhgr5h37W1Of3tzBO7IUedsVMRYNY4RFNAwslBmJ4xvUTq4_3ZKapih2hMuPNsSZbkfEZyIEykRMH3fDV_Xm&ved=2ahUKEwjjxsbigcP_AhXbnFYBHTquDeYQmxMoAnoECCQQBA), [Ralph Johnson](https://www.google.com/search?client=opera&hs=Mlq&sa=X&sxsrf=APwXEdfhyfg8b6PB5bVXCq1YN006oA9gxA:1686754391408&q=ralph+johnson&si=AMnBZoEofOODruSEFWFjdccePwMH96ZlZt3bOiKSR9t4pqlu2IeFLYdfaNAtH3GMjf60yPatdeK4Fs3IFqqfJ2ak6HUeZqK7iNG0QcHTvo3uefdLFpOz_Xbe4Agkg3lynsInUw61kpP7DXEckxsXbqJ8ZVb-2oaPux2ZvWhM3axJ6QzMTnf4JW_Hwna6Vmx9YpYJLsdof34-2eAS-g9e5On9xDaSn9Adwh4VXQO_taOD_yNWFyJPB4YZn5meuoahEd93OYKkPOZ_&ved=2ahUKEwjjxsbigcP_AhXbnFYBHTquDeYQmxMoA3oECCQQBQ) 21 tháng 10, 1994