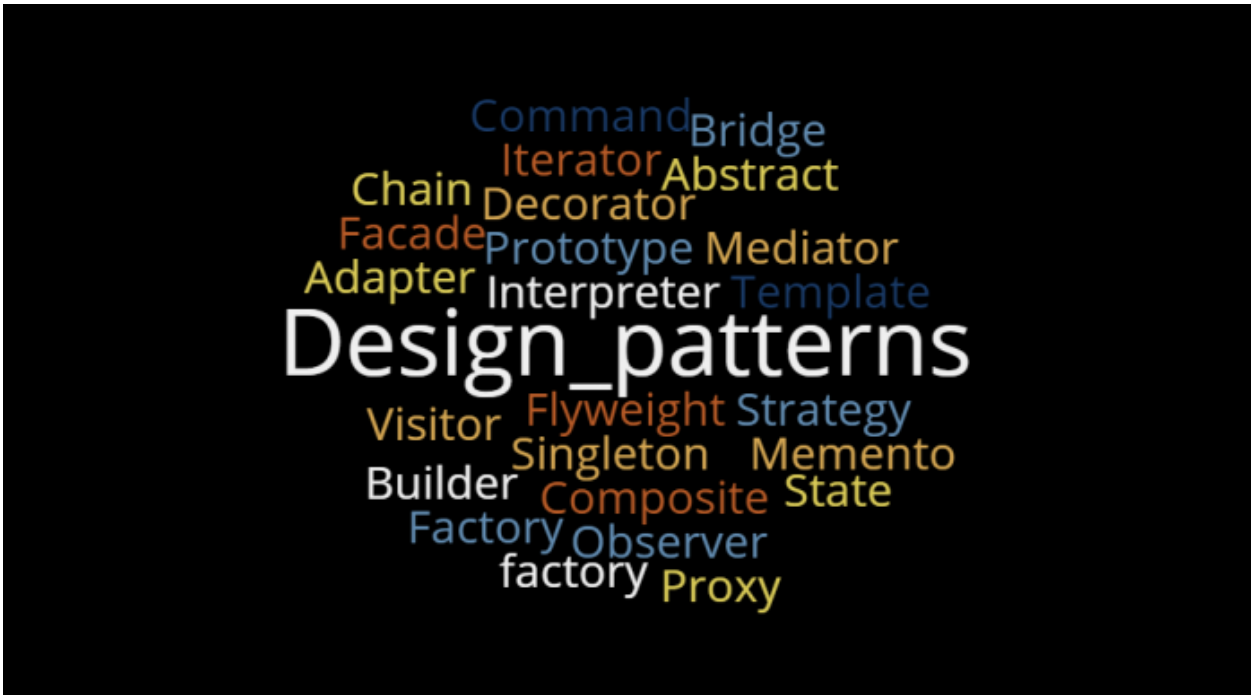
Design Pattern

(Mẫu thiết kế phần mềm)

Tác giả: Nguyễn Đình Dũng – FCT-er



I. Lời mở đầu

Chào mọi người, mình là Dũng. Trong bài viết này mình sẽ chia sẻ với mọi người về Design Pattern, hay dịch thô là Mẫu thiết kế phần mềm. Chúng ta sẽ đi qua 10 kiểu thiết kế phần mềm, phân thành 3 loại: Creational – Structural - Behavioral

Trong bài viết này mình xin phép sử dụng tiếng Anh để gọi tên các mẫu thiết kế (Creational – Structural – Behavioral) vì mình không biết dịch nó sang tiếng Việt như thế nào.

Mình chủ yếu lấy nguồn từ video của chad Fireship  :

https://www.youtube.com/watch?v=tv-_1er1mWI

và từ Refactoring.guru  :

https://refactoring.guru/design-patterns

II. Creational

Ta sẽ đi qua 4 loại thiết kế, với từng cái có tên là: Singleton và Builder.

1. Singleton

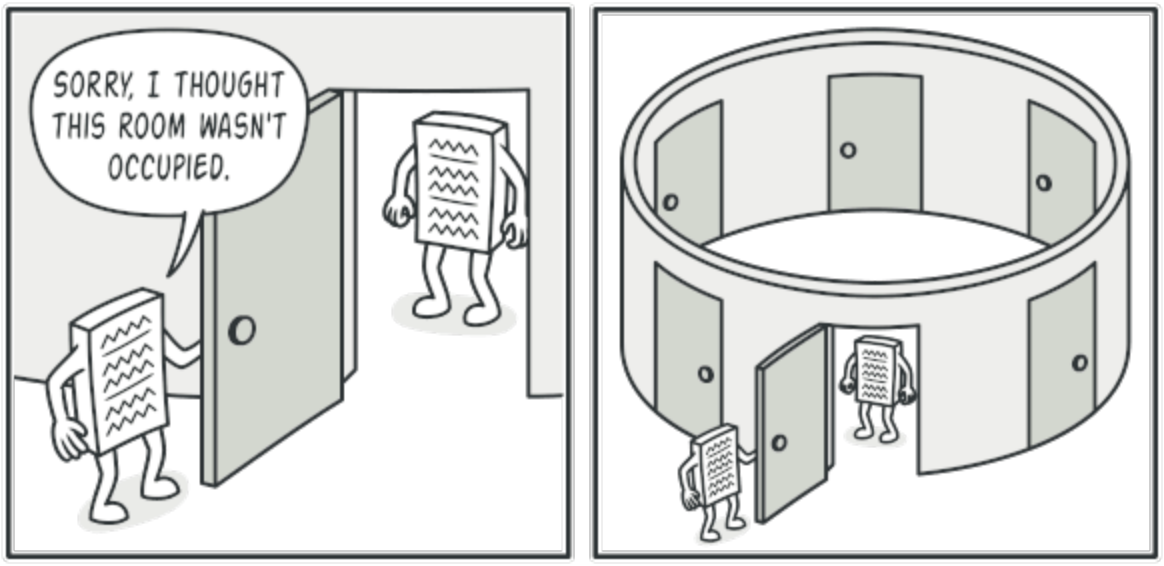
- Singleton là một loại thiết kế giúp chúng ta đảm bảo rằng mỗi class chỉ được khởi tạo 1 lần duy nhất, trong khi có thể truy cập xuyên suốt cả phần mềm

- Nó giúp chúng ta giải quyết 2 vấn đề:

    + Đảm bảo rằng chỉ có duy nhất một class làm nhiệm vụ đó được tạo ra

    + Cung cấp quyền truy cập xuyên suốt phần mềm

Nói cơ bản, nó là việc chúng ta tạo một function xử lý một công việc nào đó rồi dùng nó xuyên suốt code tổng mà thôi :>



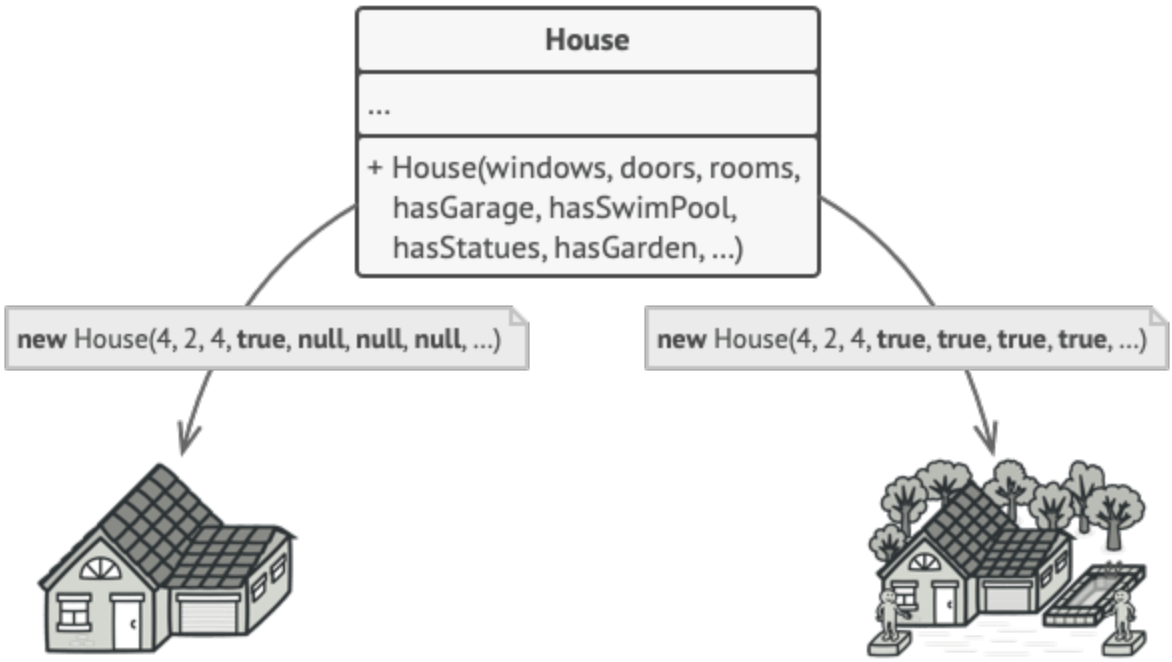
Code Python cho ví dụ của Singleton với Setting:

```python
class Setting(object):
    def __new__(cls):
        #Kiểm tra nếu như chưa được tạo setting
        if not hasattr(cls, "setting"):
            #Tạo ra một setting mới
            cls.setting = super(Setting, cls).__new__(cls)
        #Trả lại setting
        return cls.setting
setting1 = Setting()
setting2 = Setting()

print(setting1 == setting2) #True
```

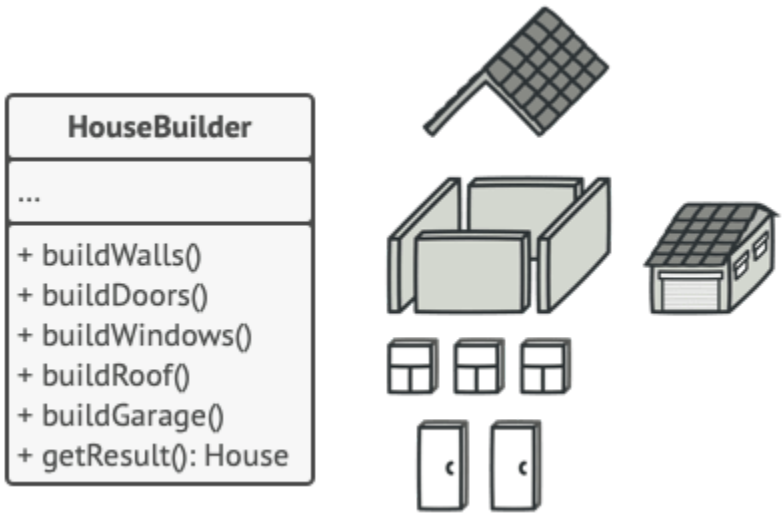
2. Builder

- Builder giúp bạn xây dựng các object phức tạp từng bước một và tạo ra các biến thể của nó một cách đơn giản

- Ví dụ như bạn muốn xây nhà, nhưng việc thay đổi, thêm thắt vào bản gốc quá khó chịu và phiền toái



Ta có thể lần lượt tạo các function để làm việc đó dễ dàng cho đôi mắt chúng ta hơn:



Code Python Builder về xây dựng một ngôi nhà:

```python
class HouseBuilder():
    def __init__(self):
        self.walls = int
        self.door = bool
        self.roof = bool
        self.windows = int
    def buildWall(self, wall):
        self.walls = wall
    def buildDoor(self):
        self.door = True
    def buildRoof(self):
        self.roof = True
    def buildWindows(self, windows)
        self.windows = windows
```

III. Structural

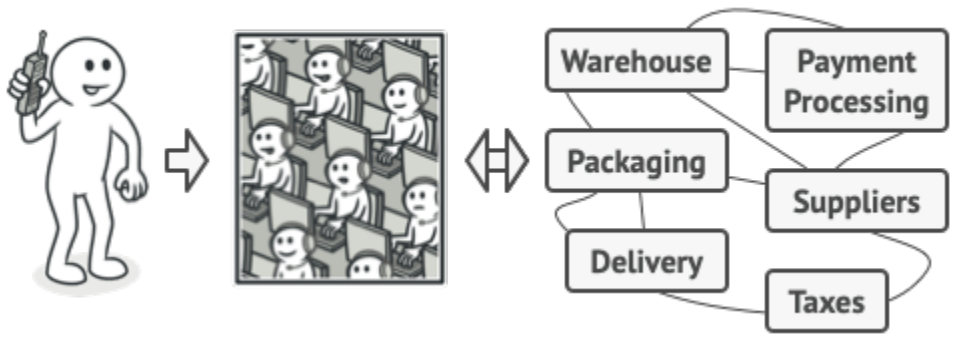
Với Structural, ta sẽ đi qua 2 loại: Façade và Decorator

1. Façade

- Façade giống như một mini-api ở trong phần mềm. Bằng cách đưa các function nhỏ vào trong một class, ta có thể đơn giản hóa việc tạo các function khác phức tạp hơn sau này, đồng thời giấu đi các function đó, giúp ta đọc code dễ hơn

- Giả sử bạn gọi một shop nào đó để đặt hàng, thì người tiếp chuyện chính là một Façade.

Họ giống như một api cơ bản, giúp bạn hoàn thành các thủ tục con như đặt đồ gì, phương thức thanh toán và biện pháp giao hàng.



Code Python cho việc order hàng online:

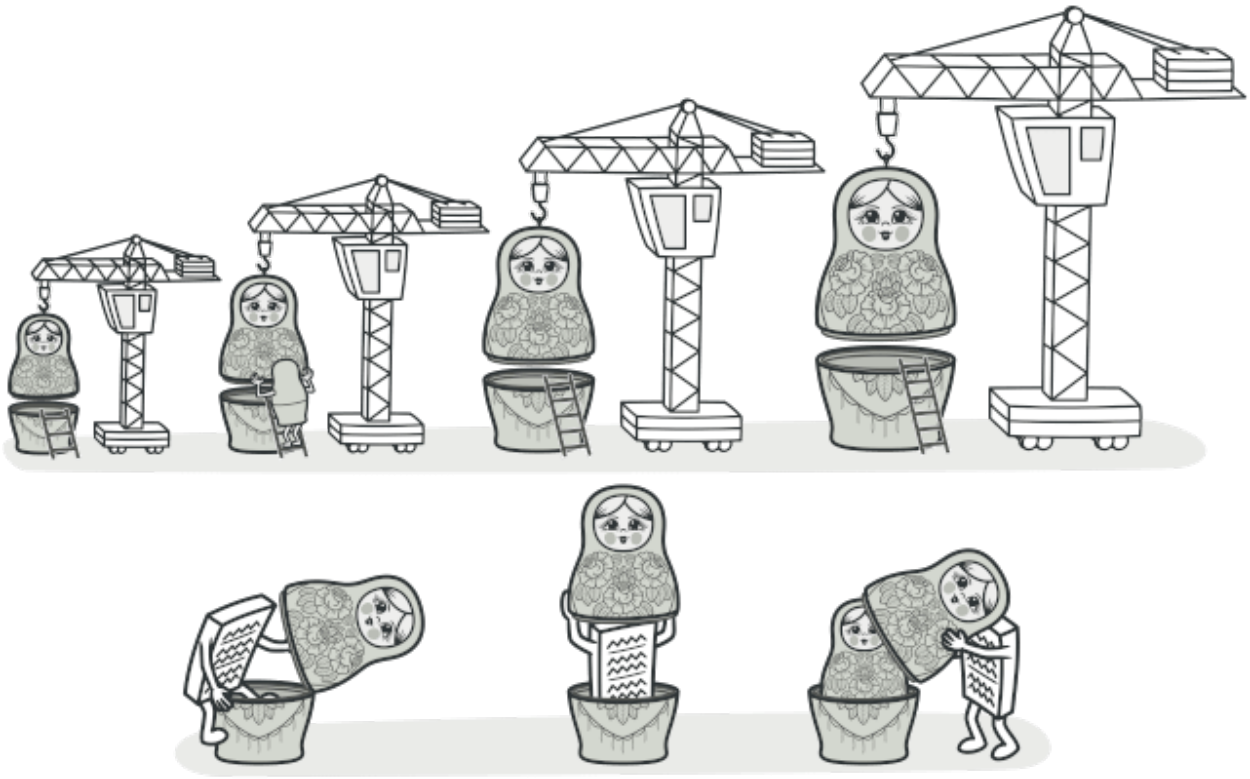
```python
class Order():
    def order(self):
        pass
class Pay():
    def pay(self):
        pass
class Trasport():
    def transport(self):
        pass

class ShopOrder():
    def __init__(self):
        self.order = Order()
        self.pay = Pay()
        self.transport = Transport()

    def OrderingStuff(self):
        self.order.order()
        self.pay.pay()
        self.transport.transport()
```

2. Decorator

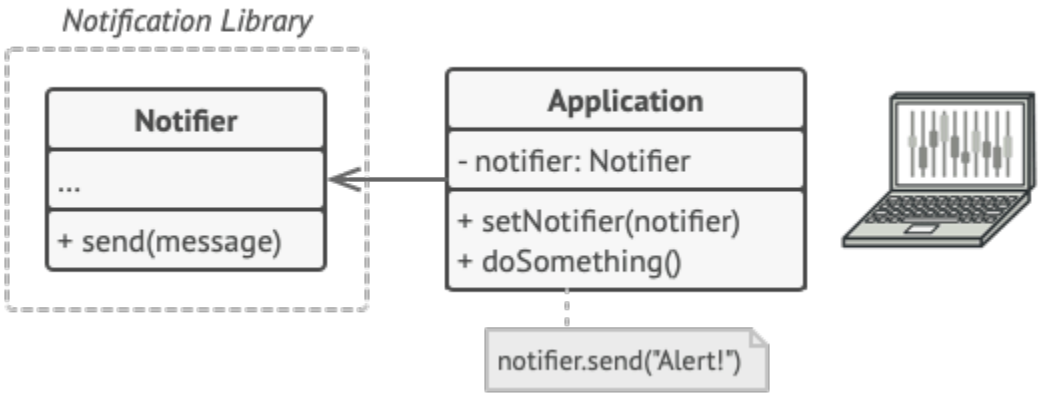
- Decorator cho phép mở rộng chức năng của một object mà không cần điều chỉnh code của nó.

- Một cách đơn giản để hiểu đó là một hàm bậc cao, nhận các hàm khác như một tham số

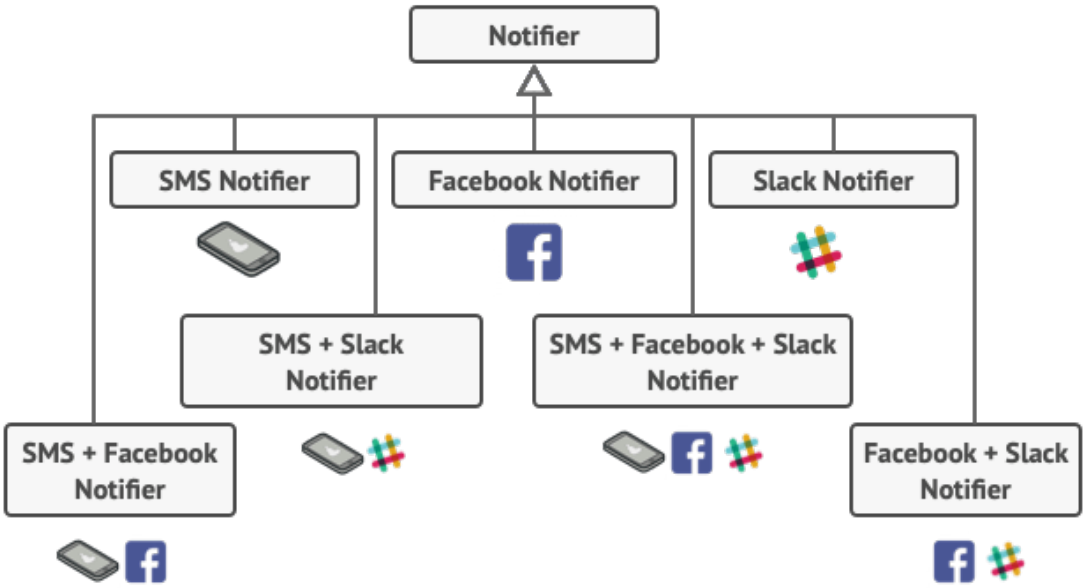
- Về cơ bản, decorator hoạt động như một wrapper, thay đổi hành vi của một đoạn code trước và sau khi một target function được thực thi mà không phải thay đổi chính target function, tăng cường chức năng ban đầu bằng cách decorating nó.



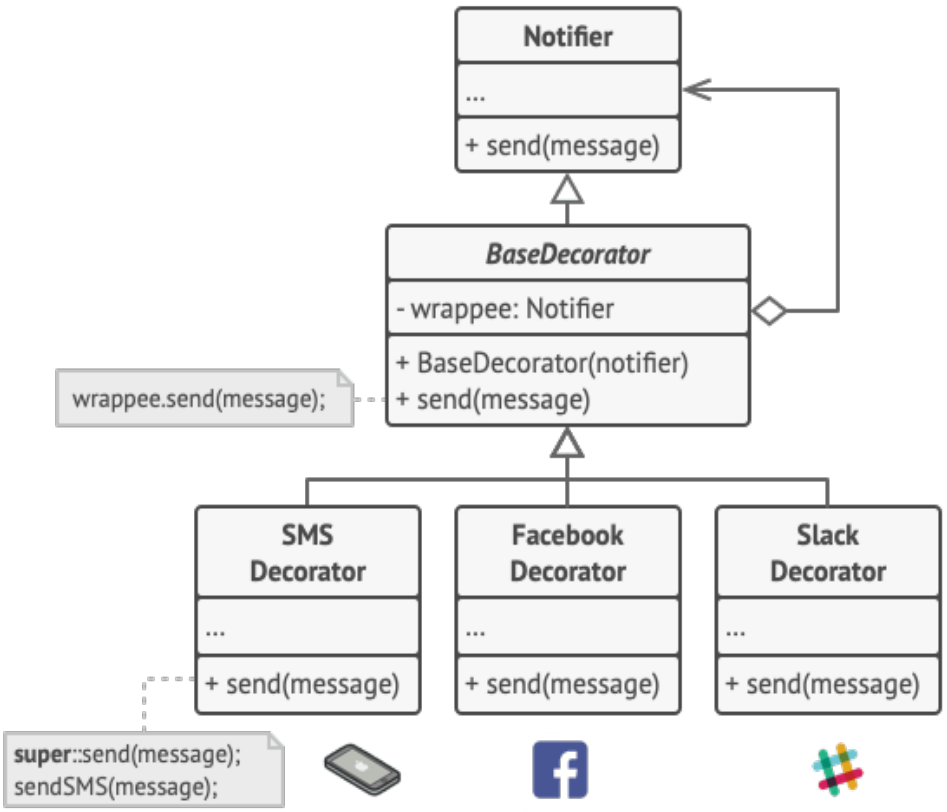
- Giả sử như bạn đang tạo ra một library chuyên cho các phần mềm thông báo các dữ liệu quan trọng tới người dùng. Lib sử dụng một class tên là Notifier để chuyển dữ liệu từ người dùng tới các email đã đăng ký.



- Tuy nhiên, khi mở rộng class ra, ta lại thấy nó rất phức tạp và tốn tài nguyên chương trình vì phải tạo ra nhiều subclass để thỏa mãn số lượng phương thức liên lạc lớn.



- Khi này, ta có thể tạo ra một rất dễ dàng Decorator như một trung gian, giúp sắp xếp lại những subclass thừa, đồng thời gửi các thông báo tuần tự như một stack. Điều này giúp giảm thiểu lượng tài nguyên phải sử dụng, đồng thời khi kết hợp các loại thông báo



Ví dụ trên trong Python:

```python
def send(message):
    pass
def SMS(message):
    send(message)
def FB(message):
    send(message)

def Decorator(func):
    def wrapper(message):
        func(message)
        print("Complete")
    return wrapper

notifier = Decorator(SMS)
notifier("New SMS")

notifier2 = Decorator(FB)
notifier2("New Facebook")
```

IV. Behavioral

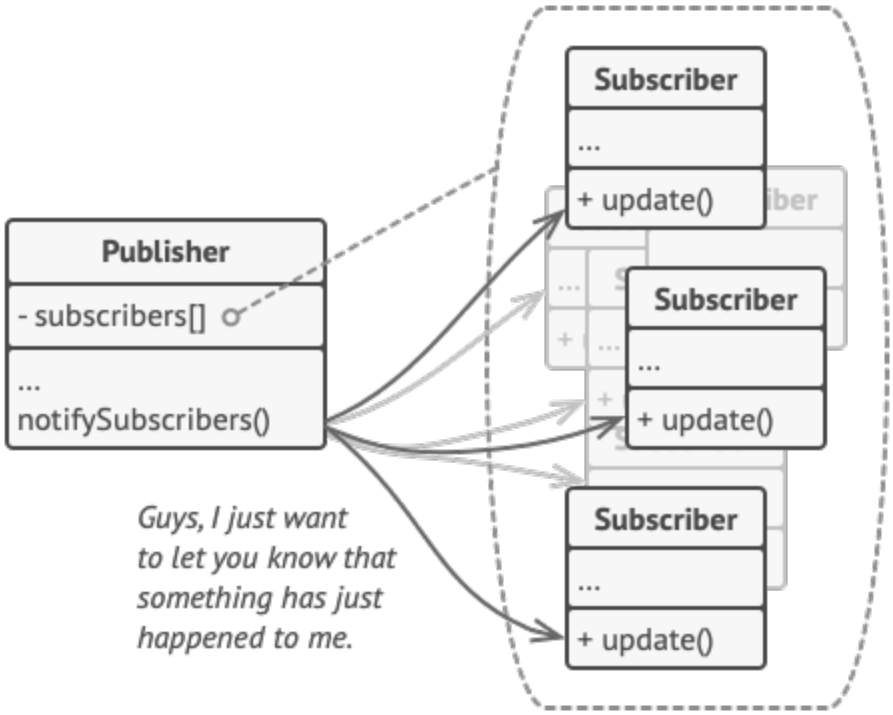
Với Behavioral, ta sẽ đi qua 2 thiết kế, bao gồm Observer và State

1. Observer

- Observer cho phép bạn tạo ra một phương thức đăng kí để thông báo một sự kiện đến với vô số object. Observer rất cần thiết cho những lập trình viên phần mềm.

- Giả sử bạn đang là chủ một cửa tiệm đồ công nghệ. Sắp tới có mẫu điện thoại mới nên có nhiều người đến cửa hàng của bạn để check xem máy đã có hàng chưa. Việc họ liên tục đến và đi để check hàng như vậy sẽ rất là vất vả. Bạn sẽ định gửi email tới tất cả mọi người xung quanh để thông báo, nhưng nó cũng sẽ gửi đi rất nhiều spam đến các khách hàng không quan tâm

- Để giải quyết vấn đề này, ta sẽ tạo ra một danh sách các subscriber, hay các khách hàng đăng kí, đồng thời tạo ra 2 function để thêm và xóa khách hàng đi. Vậy vấn đề đã được giải quyết rất nhanh chóng và gọn gàng.



Code ví dụ trong Python:

```
class Publisher():
    def __init__(self):
        self.subscriber = list
    def addSub(self, subs):
        self.subscriber.append(subs)
    def removeSub(self, subs):
        self.subscriber.remove(subs)
    def notifySub(self, func):
        func()
class Subscriber():
    def __init__(self, name):
        self.name = name
publish = Publisher()
subs1 = Subscriber("Andy")
publish.addSub(subs1)
```

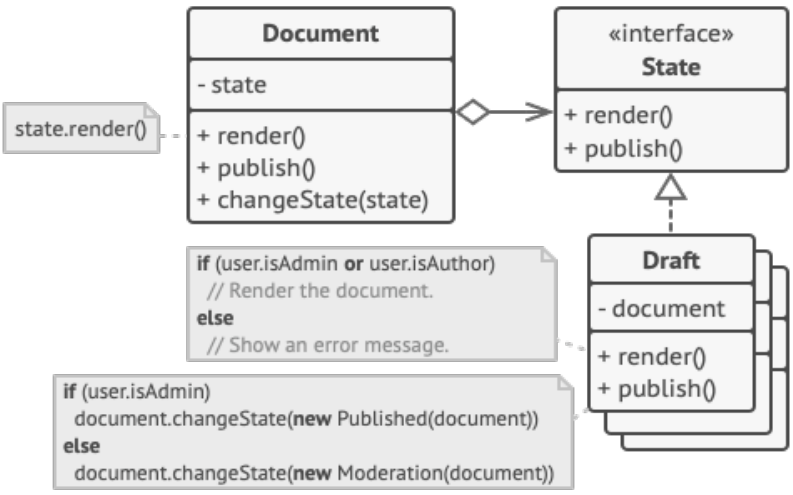
2. State

- Ý tưởng của State là bạn tạo ra vô số object nhỏ cho từng trạng thái của các object. Với mỗi trạng thái, tác dụng của object đấy sẽ thay đổi. Nó được sử dụng nếu như object cần làm nhiều việc cùng lúc trong runtime.

- State là bản cải thiện của FSM (aka "Máy trạng thái hữu hạn") khi mà FSM quyết định các trạng thái làm gì trong code của object. FSM chỉ ổn khi số lượng trạng thái nhỏ, khi lớn thì code sẽ dần bẩn đi. State thêm các object làm cho code sạch hơn, gọn hơn.

- Giả sử như bạn là một nhà báo. Khi viết xong một bài báo, nó sẽ được chuyển sang bộ phận kiểm duyệt, rồi được xuất bản.

- Khi sử dụng State, ta sẽ tạo ra tất cả các trạng thái có thể và đưa chúng vào các object độc lập. Khi thay đổi trạng thái, ta thay đổi object mang trạng thái đó bằng một object tương tự khác.



Code ví dụ trong Python:

```python
class State():
    def __init__(self, state):
        self.state = state
    def changeState(self, newState):
        self.state = newState
    def returnState(self):
        return self.state
class Document():
    def __init__(self):
        self.state = State("Draft")
    def Moderation(self):
        if self.state.returnState == "Draft":
            checkDoc()
            self.state.changeState("Modi")
        else:
            return "This was not accepted"
    def Publish(self):
        if self.state.returnState == "Modi":
            publishDoc()
            self.state.changeState("Published")
```

V. Lời kết

Bài viết này theo mình thì giống như một bản dịch thuật nửa mùa hơn là một bài viết thực thụ. Nhưng mình mong nếu ai đó lên mạng và không hiểu các Design Object là gì thì bạn có thể đọc bài viết này.