TRƯỜNG ĐẠI HỌC GIA ĐỊNH

KHOA CÔNG NGHỆ THÔNG TIN

\*\*\*\*\*

**ĐỒ ÁN MÔN HỌC**

**LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG NÂNG CAO**



**TÊN ĐỀ TÀI:**

**COMPOUND PATTERN**

|  |  |  |  |
| --- | --- | --- | --- |
| **GVHD** | : | Ths. Lê Huỳnh Phước |  |
| **SVTH** | : | Lâm Vũ Thành Tài | 2004110014–14DCPM02 |
|  | : | Trương Hoài Phong | 2004110017–14DCPM02 |
|  | : | Lê Hoàng Thạch | 2008110288–14DCPM02 |
|  | : | Trần Ngọc Anh Thái | 2004110031–14DCPM02 |

Tháng 8 – Năm 2022

**MENU**

[**1) Giới thiệu về Compound Pattern** 3](#_Toc112320918)

[**2) Ví dụ về Compound Pattern** 3](#_Toc112320919)

[**2.1 SimUDuck** 3](#_Toc112320920)

[**2.2 MVC** 15](#_Toc112320921)

[**3) Tổng kết về Compound Pattern** 23](#_Toc112320922)

# **1) Giới thiệu về Compound Pattern**

-Có thể nói chúng ta đang học về mẫu thiết kế, ngoài việc mẫu giúp chúng ta dễ bảo trì, dễ tái sử dụng code, tránh bễ hay trùng lặp code thì mỗi mẫu đều có những điểm mạnh riêng cho từng mẫu ! Nhưng trong chương trình thực tế hay một dự án lớn thì người ta không sử dụng một mẫu duy nhất mà kết hợp nhiều mẫu lại với nhau để giải quyết nhiều vấn đề cùng lúc hơn.

-Việc sử dụng từ hai mẫu trở lên trong một chương trình thì ta có thể gọi đó là Compound Pattern (mẫu ghép)

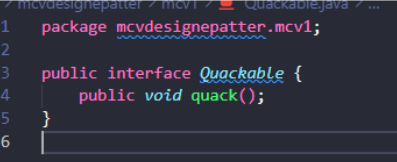
# **2) Ví dụ về Compound Pattern**

## **2.1 SimUDuck**

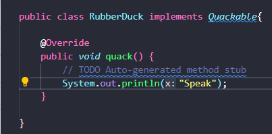
**-Adapter**

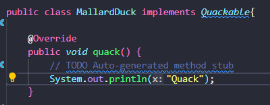
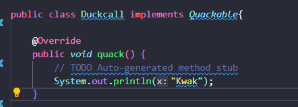
-Chương trình mô phỏng vịt SimUDuck

**-**Đầu tiên chúng ta tạo một interface Quackable có phương thức là quack() để mô phỏng tiếng kêu cho vịt



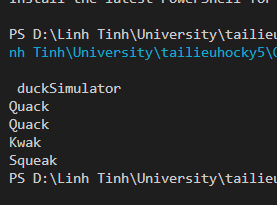
-Sau đó tạo them 4 class vịt implement Quackable





-Tuy là kế thừa phương thức quack từ Quackable những mỗi con vịt có những hành vi kêu khác nhau

-Tiếp theo chúng ta TestDrive thử:

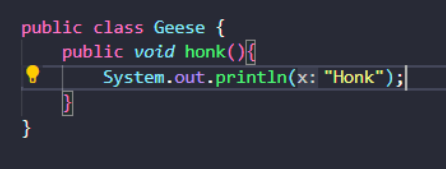
 

-Mọi thứ vẫn diễn ra bình thường, những con vịt có các hành vi kêu khác nhau đều hiển thị trên màng hình

-Nhưng một hôm khách hàng lại yêu cầu cho chúng ta rằng muốn them ngỗng vào chương trình ! Biết rằng Ngỗng thì tiếng kêu của nó khác với vịt (vịt thì kêu quack, ngỗng thì kêu honk)

-Giải pháp thường dùng là chúng ta tạo một class có tên là honkable và cho class goose implements honkable

-Nhưng có cách nào hay hơn mà không cần phải tạo them interface honker mà có thể để chuyển đổi quackable thành honkable => **adapter**



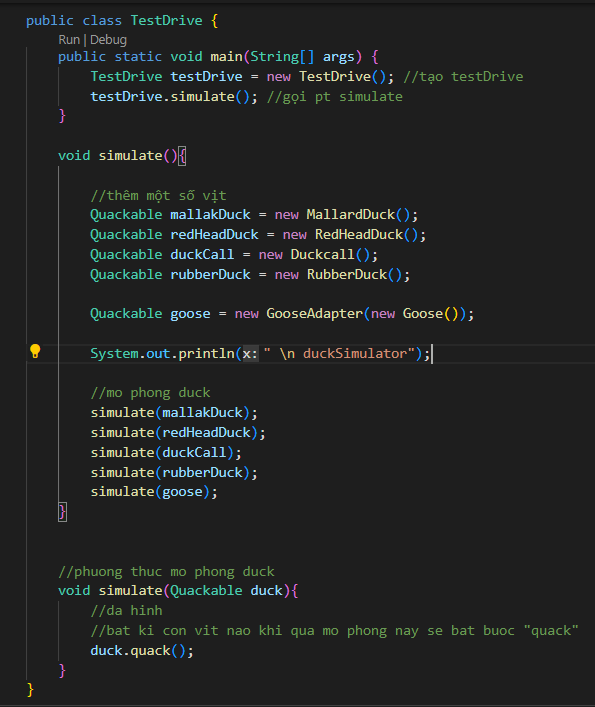
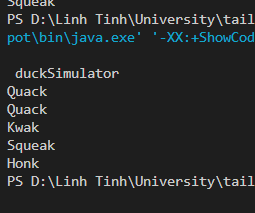
-Chúng ta tạo một class GooseAdapter và implement Quackable



-Kế thừa phương thức quack của quackable nhưng thay đổi hành vì là honk

-Khi phương thức quack() được gọi thì sẽ được ủy quyền (chuyển đổi) cho honk()

-TestDrive

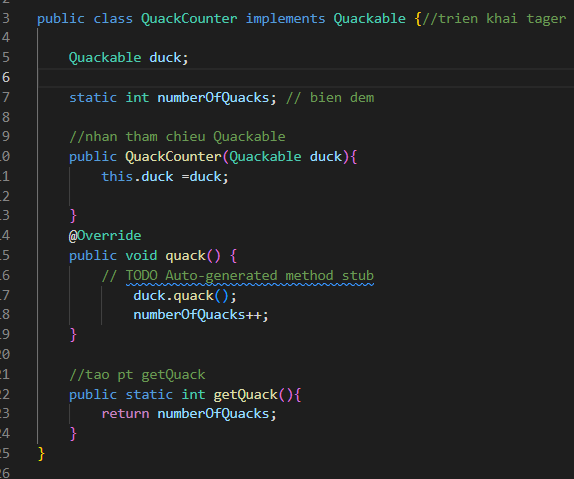
-Sau này có cần them hành vi cho ngỗng như, bơi, bay,makenoise thì chúng ta chỉ cần tạo một class chuyển đổi (adapter) và implement interface của vịt

**-Decorator**

**-**Khách hàng muốn biết tổng số lượng vịt kêu trong một đàn vịt

-Chúng ta sẽ tạo một class QuackCounter và implements Quackable

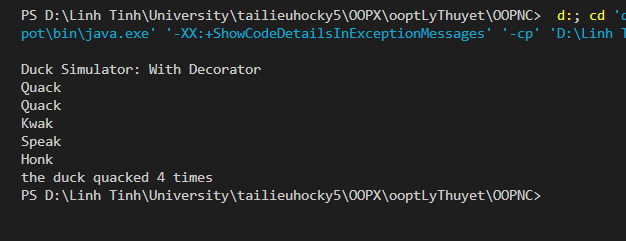
-Kế thừa quack(), và cứ mỗi lần gọi đến quack() thì biến numberofQuacks lại tang lên một.



-Tiếp theo chúng ta bọc những con vịt vào lớp decotator là QuackCounter



-Test Drive



-Những con vịt được decorate thì sẽ được đếm số lần kêu

- Ngỗng không phải là vịt và không được bọc bởi lớp QuackCounter nên không được đếm

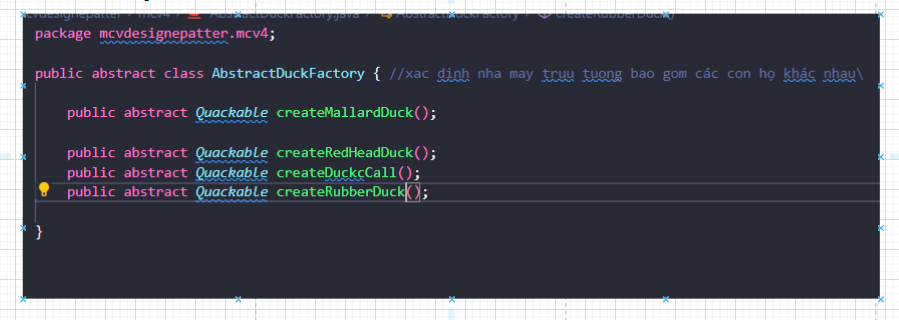
**-Factory**

Nhưng chúng tôi nhận thấy rằng quá nhiều quacks không được tính. Để giải quyết vấn đề đó chúng ta sẽ áp dụng mẫu factory.

Mới đầu ta tạo AbstractDuckFactory gồm những phương thức tạo tạo tiếng kêu cho trừu tượng cho từng con vịt.

Chúng ta cần một nhà máy để sản xuất vịt! Được rồi, chúng tôi cần một số kiểm soát chất lượng để đảm bảo vịt của chúng tôi được bọc. Chúng ta sẽ xây dựng toàn bộ một nhà máy chỉ để sản xuất chúng. Nhà máy nên sản xuất một họ sản phẩm bao gồm các loại vịt khác nhau, vì vậy chúng ta sẽ sử dụng Mô hình AbstractDuckFactory. Hãy bắt đầu với định nghĩa của lớp AbstractDuckFactory:

Chúng ta đang xác định một AbstractDuckFactory mà các lớp con sẽ thực hiện để tạo ra các họ khác nhau. Và mỗi phương thức tạo ra một loại vịt.



Sau đó ta tạo lớp DuckFactory kế thừa AbstractDuckFactory các phương thức trả về trả về con vịt .

Tiếp theo, chúng ta sẽ tạo ra một DuckFactory tạo ra những con vịt mà không có người trang trí, chỉ để có được sự kiên định của nhà máy

Mỗi phương thức tạo ra một sản phẩm: một loại Quackable cụ thể. Sản phẩm thực tế không được biết đến đối với trình mô phỏng — nó chỉ biết rằng nó đang nhận được một Quackable.



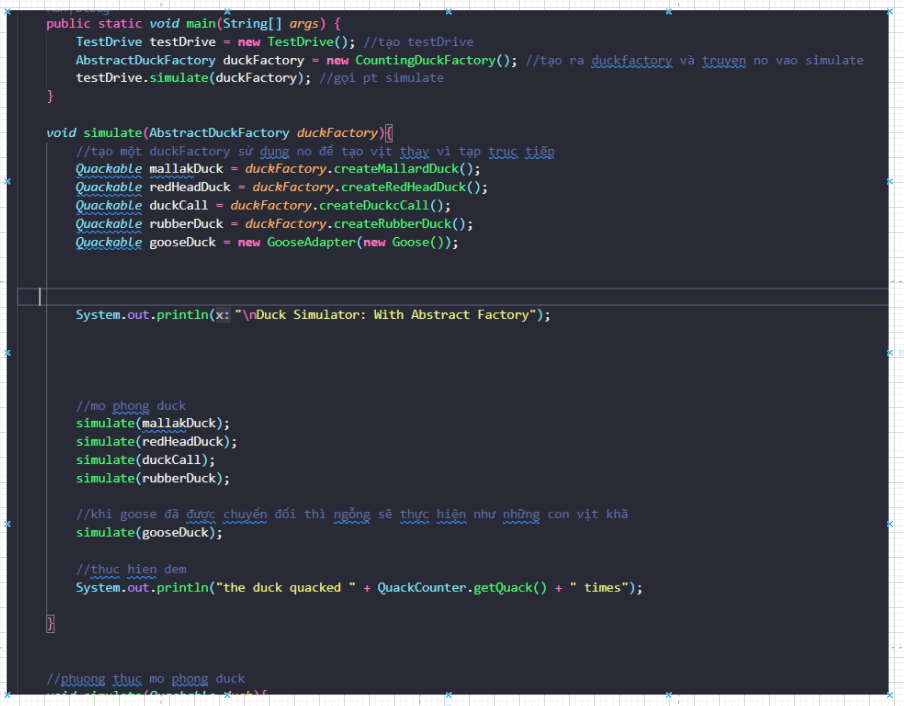
Tạo lớp Quackable dung để đếm tiếng vịt kêu nó kế thừa lại lớp AbstracDuckFactory

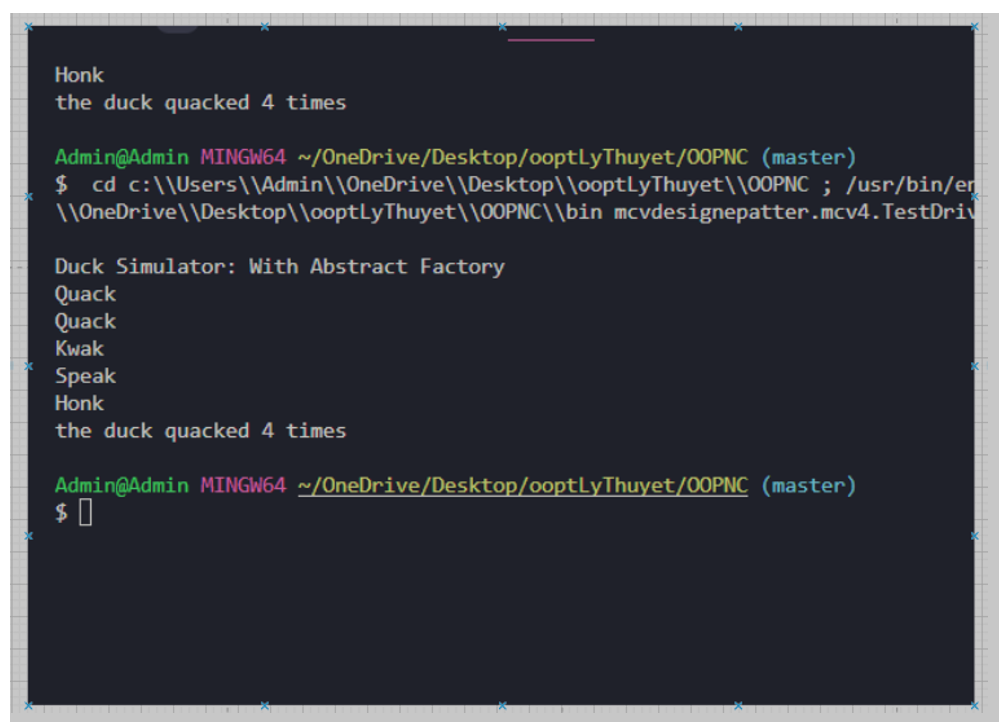
Bây giờ hãy tạo ra nhà máy mà chúng tôi thực sự muốn, CountingDuckFactory:

Mỗi phương thức bao bọc Quackable với trang trí đếm quack. Trình mô phỏng sẽ không bao giờ biết sự khác biệt; nó chỉ nhận lại một Quackable. Nhưng bây giờ các người kiểm tra của chúng ta có thể chắc chắn rằng tất cả các quacks đang được tính.



Hãy nhớ nhà máy trừu tượng hoạt động như thế nào? Chúng tôi tạo ra một phương pháp đa hình lấy một nhà máy và sử dụng nó để tạo ra các đối tượng. Bằng cách đi qua các nhà máy khác nhau, chúng tôi có thể sử dụng các dòng sản phẩm khác nhau trong phương pháp này. Chúng ta sẽ thay đổi phương thức simulate() để nó cần một factory và sử dụng nó để tạo vịt.



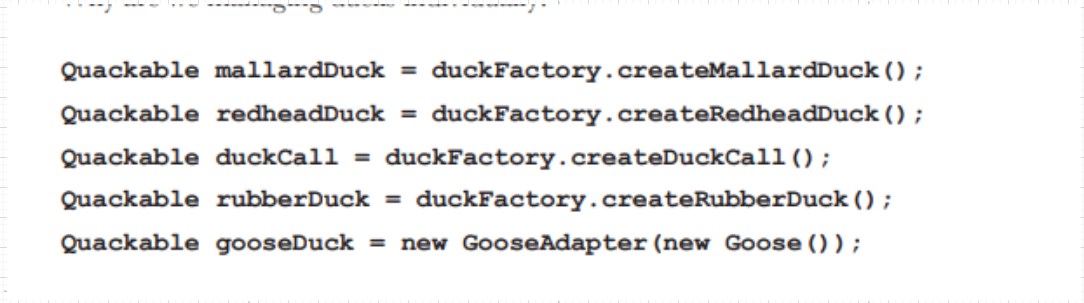


Nó trở nên hơi khó khăn để quản lý tất cả những con vịt khác nhau một cách riêng biệt. Có cách nào bạn có thể giúp chúng tôi quản lý vịt nói chung và thậm chí có thể cho phép chúng tôi quản lý một vài \"gia đình\" vịt mà chúng tôi muốn theo dõi không?

Thì composition giúp chúng ta quản lý 1 gia đình vịt 1 cách dễ dàng hơn

Tại sao chúng ta lại quản lý vịt cá nhân?

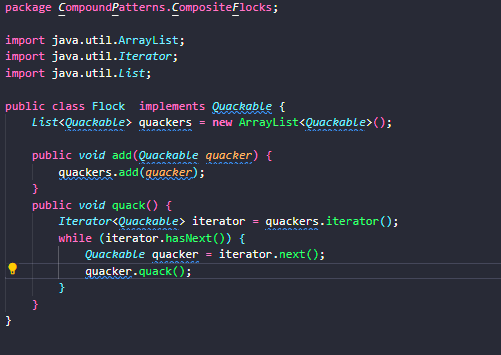
Đây là rất dễ quản lý!



Hãy nhớ Mẫu tổng hợp cho phép chúng ta xử lý một tập hợp các đối tượng theo cách tương tự như các đối tượng riêng lẻ? Còn gì tổng hợp tốt hơn một đàn Quackables! Hãy bước qua cách điều này sẽ hoạt động:

Chúng tôi sử dụng một danh sách Arraylist

**-Composite**

Phương thức add() thêm một Quackable vào Flock.

Bây giờ đối với phương thức quack() — xét cho cùng, Flock cũng là một Quackable. Phương thức quack() trong Flock cần phải hoạt động trên toàn bộ Flock. Ở đây chúng ta lặp lại qua Array

Tổ hợp của chúng tôi đã sẵn sàng; chúng ta chỉ cần một số mã để làm tròn những con vịt vào cấu trúc tổng hợp.

+Tạo tất cả các Quackables, giống như trước đây

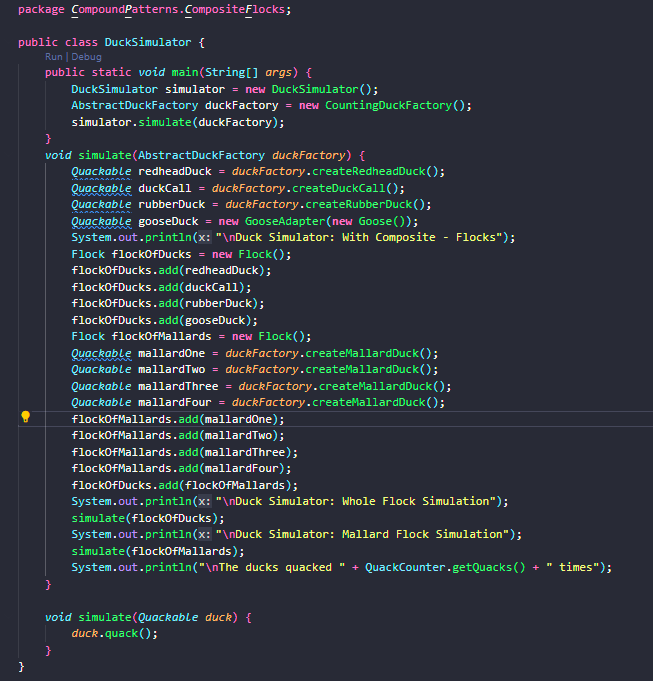
+Sau đó chúng ta tạo một Flock và tải nó lên với Quackables

+Sau đó, chúng tôi tạo ra một Đàn vịt trời mới.

+Sau đó, chúng tôi thêm Đàn vịt trời vào đàn chính.

+Hãy thử nghiệm toàn bộ Đàn chiên!

+Sau đó, chúng ta hãy thử nghiệm đàn vịt trời.

Không có gì cần phải thay đổi ở đây; một đàn chiên là một qua

**-Observer**

Bây giờ chúng ta có yêu cầu ngược lại: chúng ta cũng cần theo dõi từng con vịt. Bạn có thể cho chúng tôi một cách để theo dõi từng con vịt đang quẫy đạp trong thời gian thực không?

* Để giải quyết yêu cầu đó chúng ta sẽ áp dụng mẫu Observer.

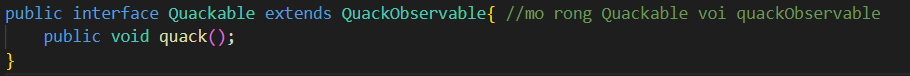
-Đầu tiên chúng ta cần một giao diện cho Subject của chúng ta có tên là Observable. Một Observable cần các phương pháp để đăng ký(, nêu ra tên đối tượng khi đối tượng đó thay đổi) và thông báo cho quan sát viên. (Chúng tôi cũng có thể có một phương pháp để loại bỏ các nhà quan sát, nhưng chúng tôi sẽ giữ cho việc triển khai đơn giản ở đây và bỏ qua điều đó.)

+ QuackObservable là giao diện mà Quackables nên triển khai nếu họ muốn được quan sát.

Text

Description automatically generated

+ Bây giờ chúng ta cần đảm bảo rằng tất cả các Quackables đều triển khai “interface”này...



-Bây giờ, chúng ta cần đảm bảo rằng tất cả các lớp cụ thể triển khai Quackable có thể xử lý là một QuackObservable.

+ Chúng ta có thể tiếp cận vấn đề này bằng cách thực hiện đăng ký và thông báo trong mỗi lớp học (như chúng ta đã tìm hiểu mẫu Observer trước đó). Nhưng lần này chúng ta sẽ làm điều đó hơi khác một chút: chúng ta sẽ gói gọn mã đăng ký và thông báo trong một lớp khác, gọi nó là Observable và soạn nó với QuackObservable. Bằng cách đó, chúng ta chỉ viết mã thực một lần và QuackObservable chỉ cần đủ code để ủy quyền cho lớp trợ giúp Observable.

+ Hãy bắt đầu với lớp trợ giúp Observable.



++ Observable thực hiện tất cả các chức năng mà một Quackable cần phải có thể quan sát được. Chúng ta chỉ cần cắm nó vào một lớp và có đại biểu lớp đó cho Observable.

++ Observable phải implement QuackObservable vì đây là những cách gọi phương thức tương tự sẽ được ủy quyền cho nó.

++ Trong constructor, chúng ta được truyền qua QuackObservable đang sử dụng đối tượng này để quản lý hành vi có thể quan sát được của nó. Kiểm tra phương thức notifyObservers() bên dưới; bạn sẽ thấy rằng khi một thông báo xảy ra, Observable sẽ chuyển đối tượng này dọc theo để người quan sát(observer(người đăng ký)) biết đối tượng nào đang kêu.

-Bây giờ hãy xem cách một lớp Quackable sử dụng trình trợ giúp này... Tích hợp trình trợ giúp Observable với các lớp Quackable.

+ Tất cả những gì chúng ta cần làm là đảm bảo rằng các lớp Quackable được implement(sáng tác) với một Observable và họ biết cách ủy thác cho nó. Sau đó, họ đã sẵn sàng để trở thành Observables. Đây là class MallardDuck; những con vịt khác cũng tương tự như MallardDuck.

Text

Description automatically generated\

-Tiếp theo chúng ta chỉ cần làm việc ở phía Observer của mô hình.

+ Chúng ta implement mọi thứ chúng ta cần cho Observables. Bây giờ chúng ta cần một số Observer. Chúng ta sẽ bắt đầu với interface Observer:

++ interface Observer chỉ có một phương thức, update(), được thông qua QuackObservable đang quacking.

Graphical user interface, text

Description automatically generated

Bây giờ chúng ta cần một Observer: Quackologist.

Text

Description automatically generated

Chúng ta cần triển khai interface Observer nếu không chúng ta sẽ không thể đăng ký với QuackObservable. Quackologist rất đơn giản; nó chỉ có một phương pháp, cập nhật (), in ra Quackable vừa quacked.

-Chúng tôi đã sẵn sàng để quan sát. Hãy cập nhật trình mô phỏng và dùng thử:

Text

Description automatically generated

Tất cả những gì chúng tôi làm ở đây là tạo ra một Nhà nghiên cứu quackologist và đặt anh ta làm người quan sát đàn chiên.

Text

Description automatically generated

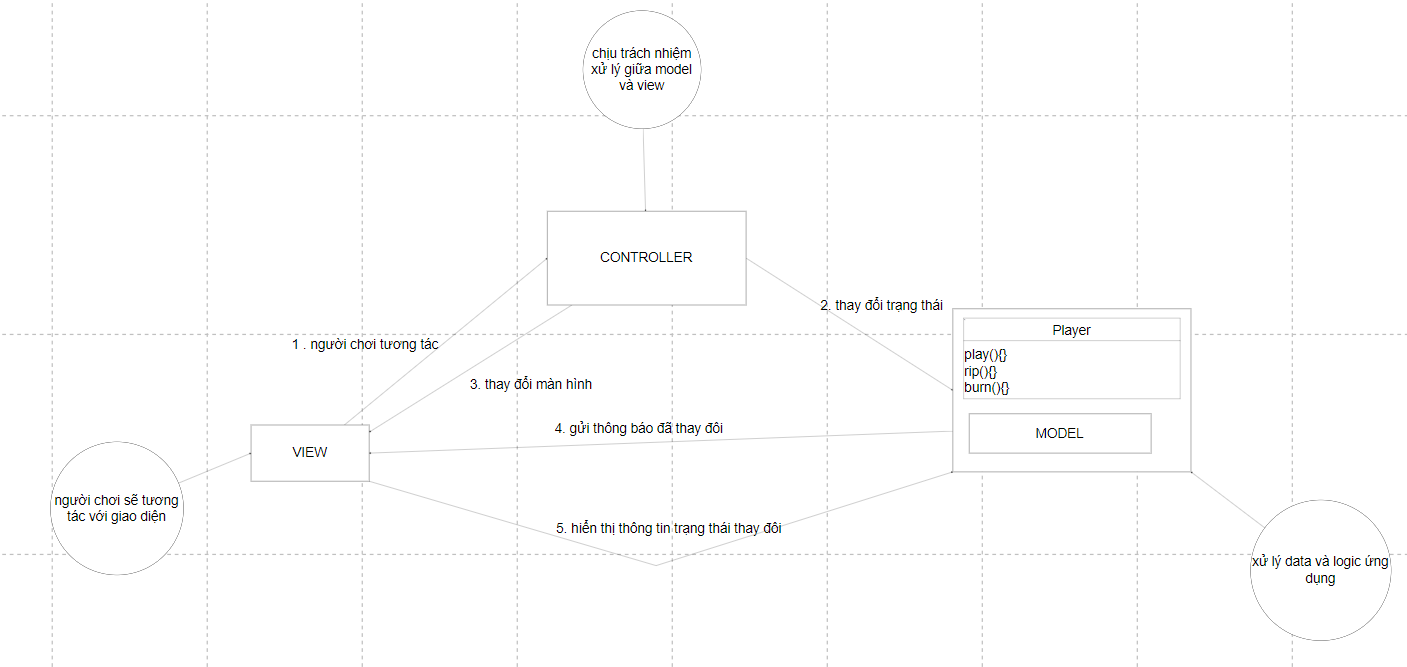
Diagram, schematic

Description automatically generated

## **2.2 MVC**

MVC Bao gồm 3 thành phần chính

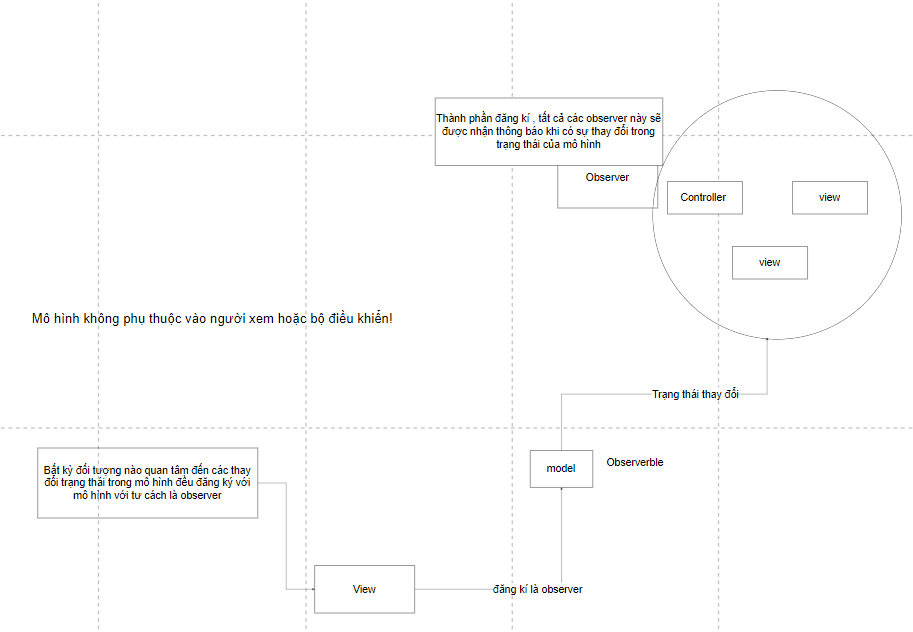
* **Model**: nơi chứa tất cả dữ liệu , trạng thái và logic . Nó cung cấp một interface để thao tác và truy xuất trạng thái và gửi thông báo về tất cả những thay đổi về cho observer.
* **Controller** : nhận Input của người dùng , xử lý sự kiện thích hợp đối với model.
* **View :** nhận trạng thái và dữ liệu cần hiển thị trực tiếp từ model

****

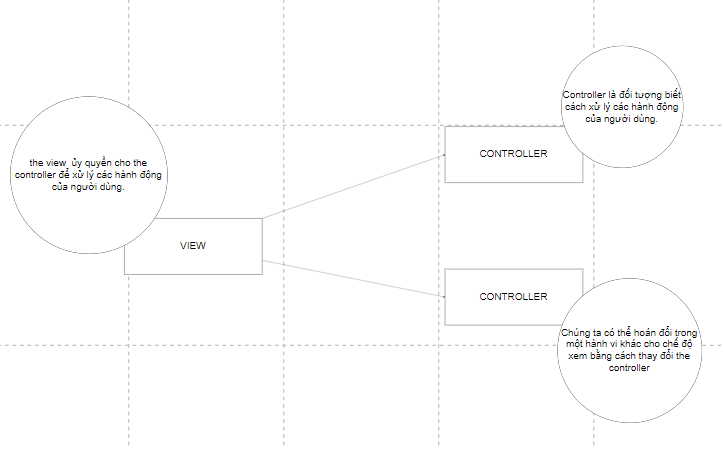
MVC như một tập hợp các mẫu thiết kế hoạt động cùng nhau.

* Về View : bao gồm các cửa sổ , button, ô nhập, văn bản…. mỗi thành phần này là một composite khi mà controller yêu cầu view cập nhật , view chỉ cần thông báo cho component và Composite xử lý phần còn lại
* Model : áp dụng observer để giữ cho các đối tượng được thông báo khi thay đổi trạng thái xãy ra . Sử dụng Observer giữ cho model hoàn toàn độc lập với view và controller, điều này cho phép ta sử dụng nhiều view khác nhau trong khi ta chỉ sử dụng duy nhất một Model

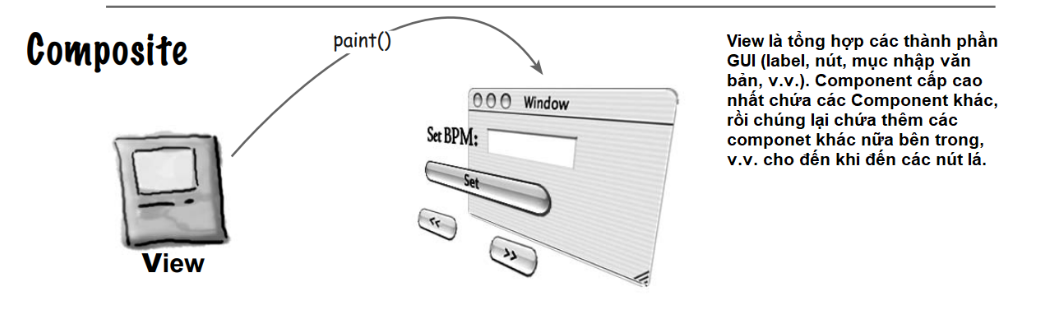
Observer



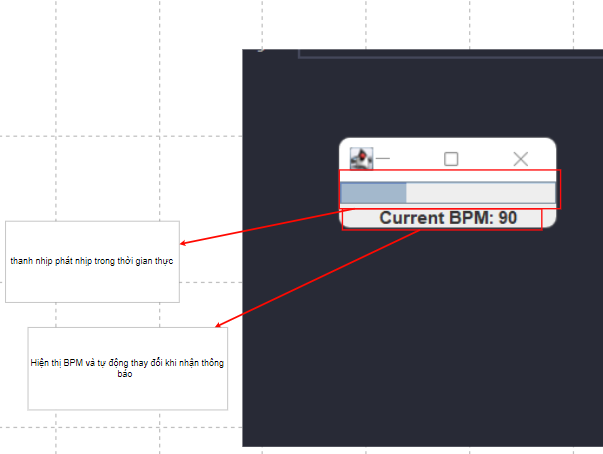
Strately

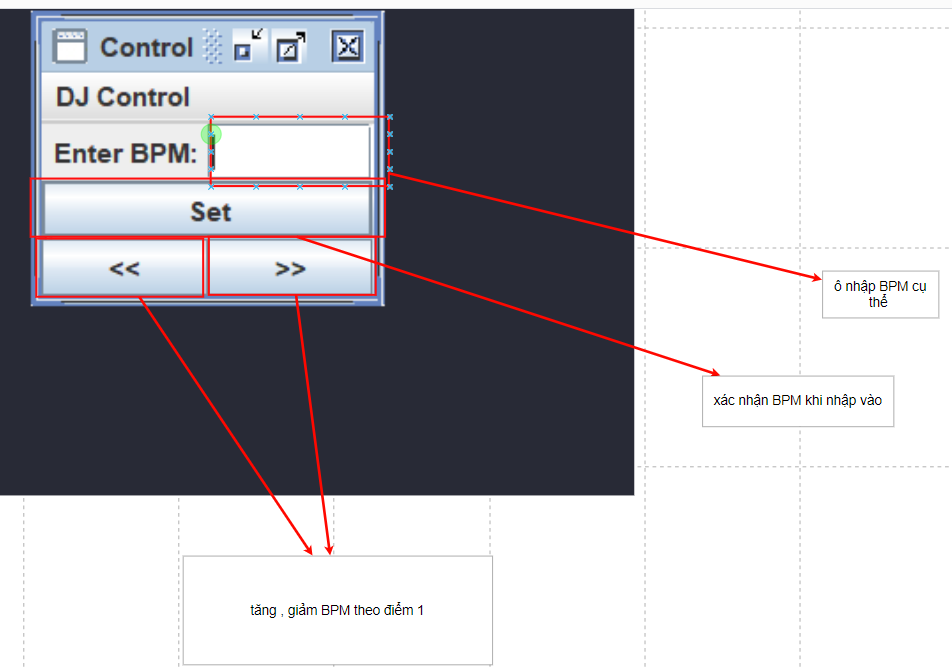


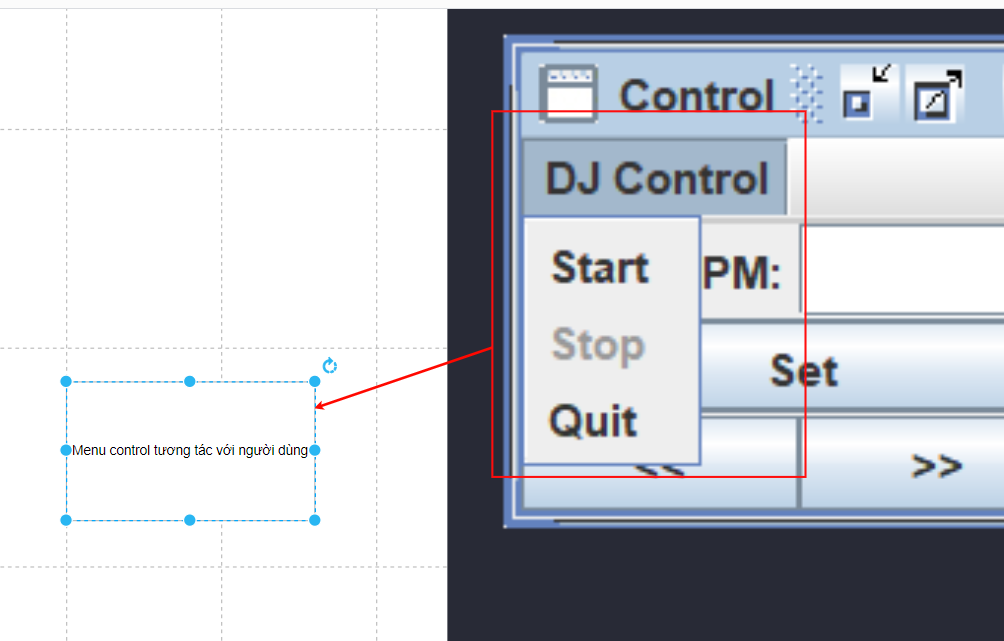
Composite



Ví dụ về Djview







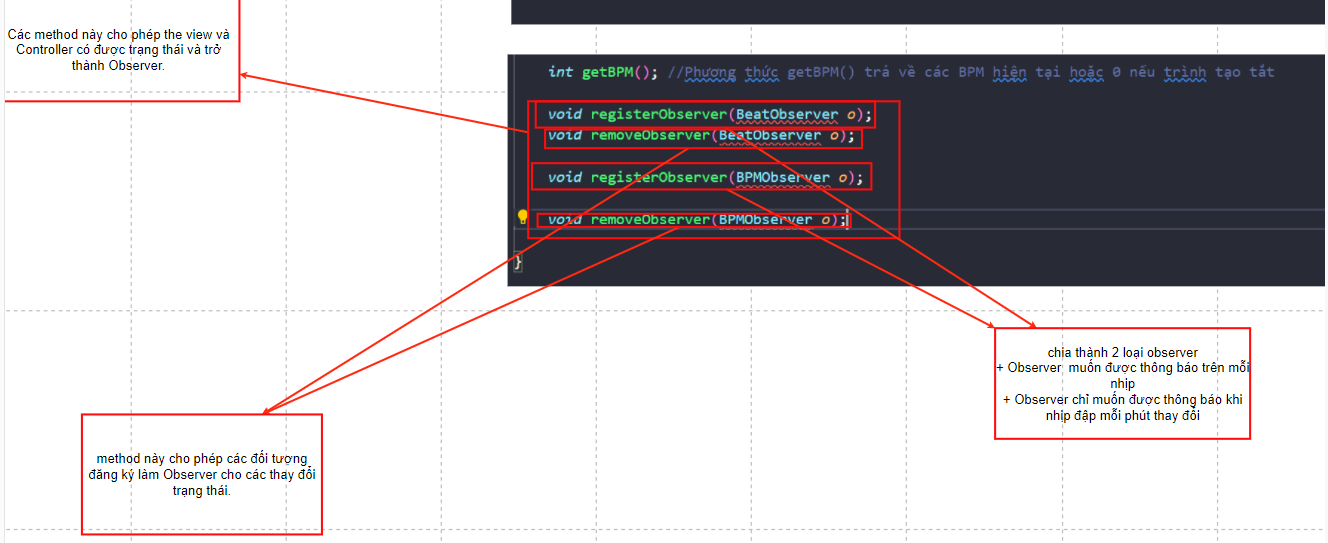
Tất cả các hành động của người dùng được gửi đến controller.

Controller sẽ lấy input từ người dùng và tìm ra cách dịch nó thành các yêu cầu trên model

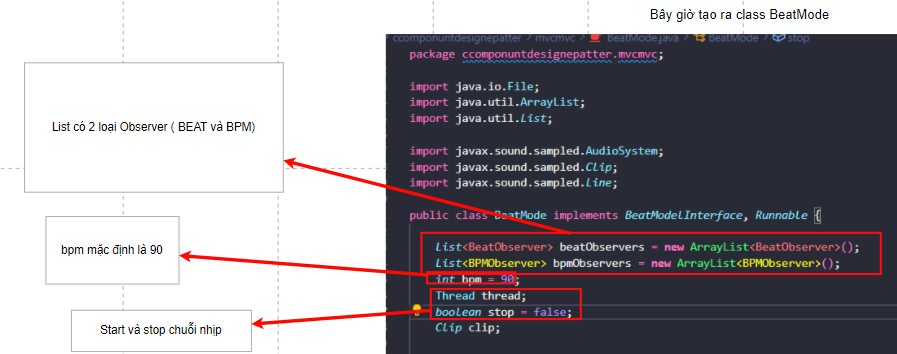


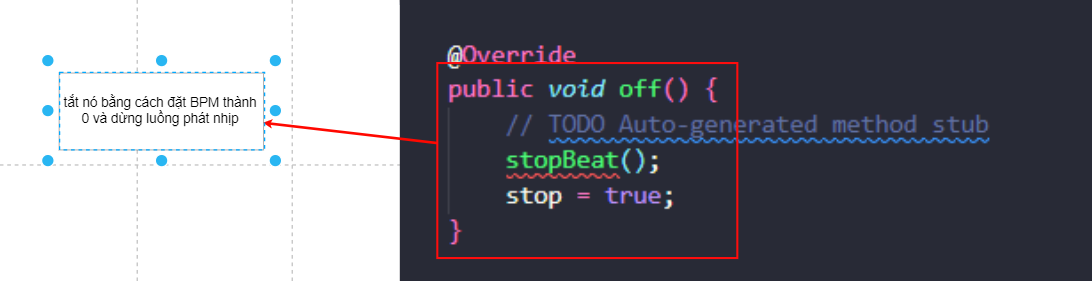
MODEL chịu trách nhiệm duy trì tất cả dữ liệu, trạng thái và bất kỳ logic ứng dụng nào. Vậy BeatModel có tác dụng gì trong đó? Công việc chính của nó là quản lý nhịp, vì vậy nó có trạng thái duy trì nhịp hiện tại mỗi phút và mã để phát một đoạn âm thanh để tạo nhịp mà chúng ta nghe thấy. Nó cũng hiển thị một giao diện cho phép bộ điều khiển thao tác với nhịp và cho phép chế độ xem và bộ điều khiển có được trạng thái của MoDEL. Ngoài ra, model sử dụng Observer Pattern, vì vậy  ta cũng cần một số phương pháp để cho phép các đối tượng đăng ký làm observer và gửi thông báo.



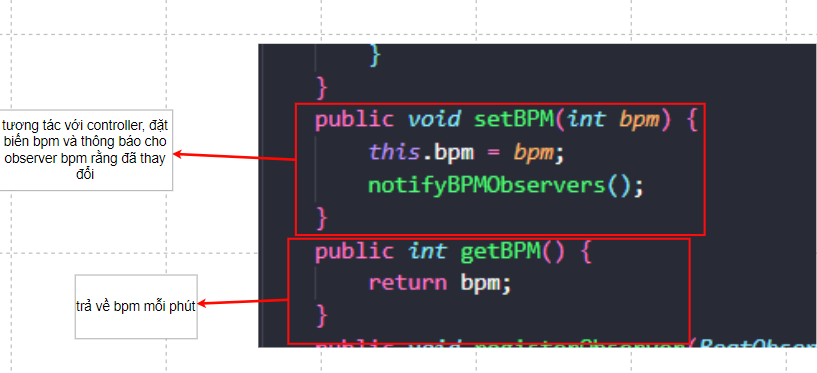


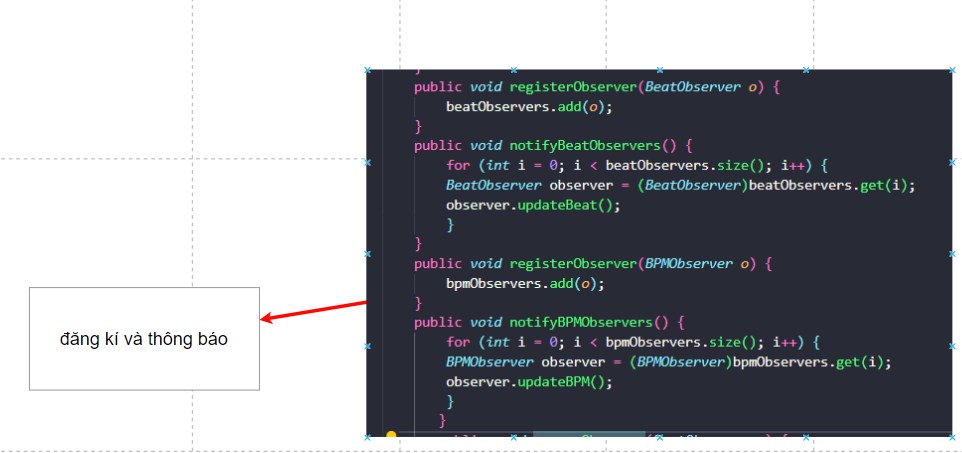
Tạo class BeatModel





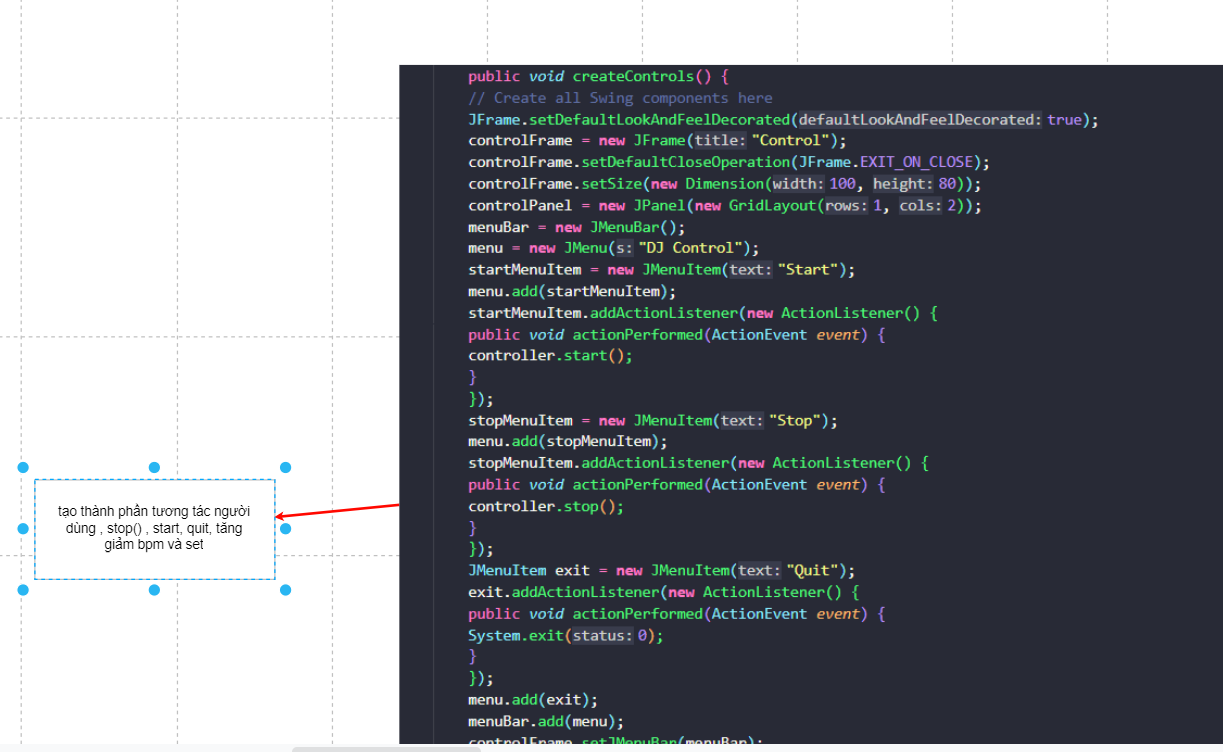


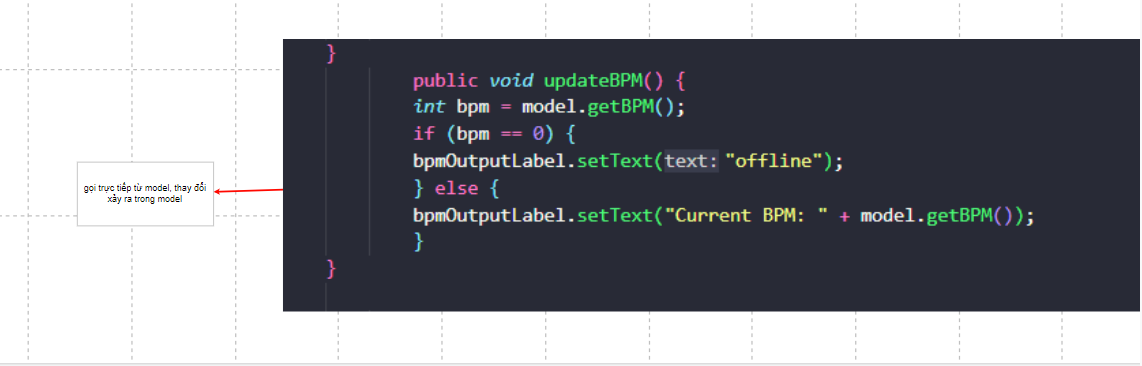




Tạo class DJView







Tạo class interface controllerInterFace



Tạo class BeatController Implements controllerInterFace

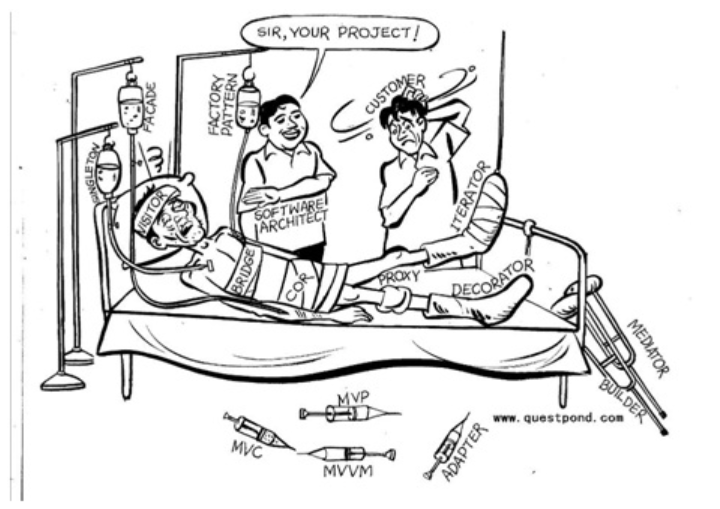


# **3) Tổng kết về Compound Pattern**

**-**Có thể thấy khi kết hợp nhiều mẫu thì chúng ta giải quyết được nhiều vấn đề cùng lúc ! Nhưng không vì thế mà chúng ta lạm dụng các mẫu một cách vô tội vạ.

-Cần phải phân tích dự án, chương trình của mình trước xem có đang gặp phải vấn đề gì và cần sử dụng các mẫu nào để giải quyết vấn đề đó cho phù hợp.

-Vì khi áp dụng mẫu một cách bừa bãi sẽ khiến chương trình chúng ta ngày càng phức tạp, nhẹ thì dẫn đến khó bảo trì nặng thì phải đập cả dự án!

****