## Introduction

Trong bài viết này chúng ta sẽ học cách thực thi design pattern và architecture pattern sử dụng C#. Chúng ta sẽ không đi vào từng pattern một nhưng xa hơn chúng ta dùng a sample project và thử thực thi mọi thứ trong nó.

Vậy tại sao bài này chúng ta tiếp cận bằng việc dựa trên a project mà không phải là tiếp cận dựa trên một ví dụ.

Design pattern và architecture patterns là quá trình suy nghĩ. Quá trình suy nghĩ không thể được giải thích bởi PPT, UML diagrams etc. Bạn cần nhìn code, bạn cần cảm thấy nó, móc nối nó với a real project scenario.

Nếu bạn nhìn vào phần lớn các bài viết về design pattern trên internet/books thì hoặc là được giải thích với chỉ UML diagrams (Không phải tất cả các developer hiểu về UML) hoặc các ví dụ giống như Car, Tree, Huma’s etc tất cả những cái này không làm cho bạn cảm thấy là những ví dụ thực tế.

Vì vậy chúng ta bắt đầu với một sample project requirement và bắt đầu việc code hoặc designing một application và cho phép design patterns ngã vào.

Myth 1: Một kiến trúc tốn phải sử dụng tất cả các design pattern trong một project.

Fact: Patterns đi ra một cách tự nhiên và hoàn toàn là theo yêu cầu.

## Design pattern VS Architecture pattern VS Architecture Style

Myth2: Design patterns và architecture patterns là giống nhau.

Fact: Design patternare là một mã giả thấp nhất level trong khi architecture patterns là tại component level.

Nghĩa của từ “Pseudo”:- Approximately it looks like that.

Chúng ta đã thấy nhiều người sử dụng các từ vựng thay thế cho nhau. Nhưng có sự khác biệt đáng kể trong cái cách mà họ làm việc.

Đầu tiên chúng ta thử hiểu về từ “patterns” và sau đó chúng ta sẽ chia nó sâu hơn.

Nếu bạn nhìn vào giải thích nghĩa của từ “pattern” trong tiếng ánh:- They are recurring and predictable events.

Cho ví dụ thì khí hậu thay đổi theo a pattern. Đại khái thì ta có mùa hè thì tiếp theo là mưa và sau đó lạnh. Con người nhận định các patterns này để tổ chức cho chính họ một cách tốt hơn.

Cùng cách đó trong thế giới software các vấn đề cái mà thường xuyên gặp là một pattern xác định và các developers đã giải quyết các vấn đề này và cho ra với một solution. Muộn hơn thì các solutions này đã chứng minh giá trị của chúng trong một khoảng thời gian và đã trở thành standard solution cho các vấn đề pattern này.

Cho ví dụ nếu chúng ta muốn SORT sau đó chúng ta đã thử nghiệm các thuật toán như bubble sort, inserted sort etc.

Design Pattern là pseudo code level solutions trong khi architecture pattern là 30000 feet level solutions đã định nghĩa tại component level. Hiểu đơn giản, nếu ai đó nói “X” là một design pattern, Expect code, nếu ai đó nói “Y” là một Architecture Pattern expect some kind of component level block diagram.

Architecture Style là một quá trình tư duy, một nguyên tắc cái mà chỉ đi kèm trong một lớp lót. Ví dụ REST là một architecture style nơi mà chúng ta trao tầm quan trọng tới HTTP

Bên dưới là ví dụ cho một trong số dhungs

Design pattern Factory, Iterator, Singleton

Architecture Pattern MVC, MVP, MVVM

Architecture Style REST, SOA, IOC

## Design pattern definition

Trước khi chúng ta đi tới project, thử đặt một định nghĩa xung quanh design pattern và sau đó chúng ta sẽ define architecture pattern

Nếu nhìn vào định nghĩa chính thức cho design patternm nó đi tới những thứ sau:

*“Time tested Solution for recurring architecture problems”*

*“Solution cho các vấn đề kiến trúc định kì đã qua kiểm tra”*

Nhưng thẳng thắn mà nói thì định nghĩa này sẽ không đúng nếu ta bắt đầu thực thi và đọc design pattern. Tại mỗi thời điểm ta sẽ nhìn design pattern là tất cả các nguyên tắc OOP tốt.

Vì vậy hãy để tôi đặt định nghĩa của tôi theo sự hiểu biết và kinh nghiệm của tôi. Tôi có thể bảo đảm rằng khi chúng ta bắt đầu thông qua tất cả desgin patterns định nghĩa này sẽ clear hơn.

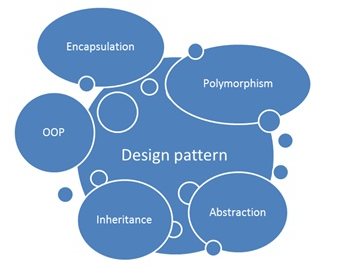
*“Time tested Solution for recurring OOP problems”*

*“Solution cho các vấn đề OOP định kì đã kiểm tra theo thời gian”*

Myth 3:- design pattern makes you an complete architect

Reality:- design pattern is one of the things forr an architect. It make you a better OOP

Thực chất DP chỉ làm bạn trở nên tốt hơn trong OOP cái là một trong rất nhiều khía cạnh để trở thành an architect. Nhưng nó không chỉ là cái để trở thanh architect. Chúng ta cũng có thể nói DP là một cách để hiểu OOP với scenarios.



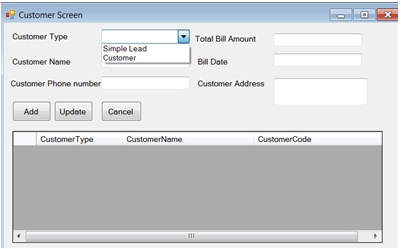
## The Cool Shop Project: - Phase 1

“Cool Shop” là một trung tâm mua sắm bán lẻ lớn. The company management muốn một hệ thống quản lý khách hàng đơn giản cho các cửa hàng bán lẻ của họ với các tính năng sau.

Company đã quyết định khởi động the project trong nhiều phases.

Trong phase 1, họ chỉ muốn nắm bắt được thông tin khách hàng. Bên dưới là các yêu cầu về thông in chi tiết

1. App sẽ có 5 trường Customer Name, Phone number, Bill Amount, Bill date and Customer Address
2. Trong phase 1 có 2 kiểu dữ liệu của customer được thu thập. Một là lead và cái khác là một customer. Một lead là một người đến cửa hàng nhưng không làm bất cứ thứ gì. Anh ta chỉ điều tra và biến mất. Một customer là một người đến và mua nhiều thứ ở của hàng. Một customer thực chất làm một giao dịch tài chính.
3. Khi nó là lead thì chỉ có Customer name và phone number là bắt buộc nhưng với một Customer tất cả các trường thì là bắt buộc. System sẽ có một điều khoản để thêm new validation rules một cách liên tục và các validation rules sẽ có thể linh hoạt và tái sử dụng để apply tới system.
4. System sẽ có khả năng để display, add, update and delete customer dât.
5. System sử dụng SQL Server và ADO.NET as the data layer technology. Nhưng trong những tháng tới chúng ta có thể migrating this data layer to EF. Migration sẽ liên tục liền mạch mà không có nhiều thay đổi trên toàn hệ thống.
6. System sẽ có khả năng cancelling any modification on the screen. Vì vậy nếu custommer đang edit một record và anh ta đã thay đổi các giá trị, anh ta sẽ có cơ hội để revert lại các giá trị cũ.



## Software Architecture is an evolution

Myth 4:- Architecture should be perfect and right at the first time.

Reality:- Architecture is evolution. Start small and then kêp improving along the way

Phần lớn mọi người tử học cách design và architecture pattern bằng cách đi qua code sau đó code và pattern sau đó pattern. Cách này rất có hại vì bãn sẽ chỉ nhìn thấy code trên lý thuyết nhưng nó không bao giờ trở thành một phần tự nhiên của ta.

Cách tốt nhất để học DP là để nhìn thấy full quá trình phát triển và làm một project và áp dụng patterns một cách tự nhiên và dần dần.

## What is an ENTITY

Đầu tiên chúng ta cần làm là xác định entities. Vì vậy ta cần hiểu chính xác nghĩa của entities trong tiếng anh.

Entities là các cái mà bạn nhìn trong thế giới thực tế của bạn. Những thứ này có thể được xác định là duy nhất. Cho ví dụ a person, place, things etc là những ví dụ của entities.

Trong thế giới OOP the technical name to entities is objects. Vì vậy trong bài này ta sử dụng các từ này để thay thế cho nhau.

## Step1:- Identify your entities/ objects

Thought process:- Software code phản ánh thế giới thực, người thực, Vì vậy nếu a accountant trong thế giới thực thì trong your software code sẽ có an entity called as accountant.

Software applications thì được tạo để tự động hóa real word and real people. Vì vậy nếu your software lặp lại real world objects bạn có thể quản lý your software tốt hơn.

OOP says that your code should reflect your domain behavior and domain entities.

Domain means the business unit, its rules and how it works.

Vì vậy bước đầu tiên là để nhận định entity/ objects từ các yêu cầu trên. Từ các yêu cầu phía trên thì dưới đây là các identifued nouns:

* Mumbai
* Pune
* Customer
* Lead
* Cool Shop

Thought process 2:- Nouns sẽ trở thành entities và verbs trở thành actions for entities. Pronoun trở thành properties và behavior cho các entities này.

# Behavioral design pattern

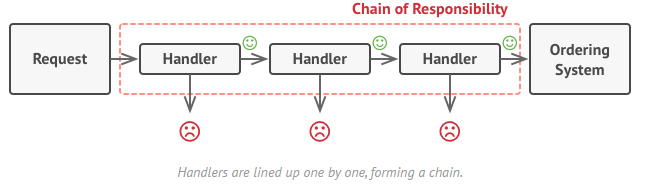
## Chain of Responsibility

### Mục đích

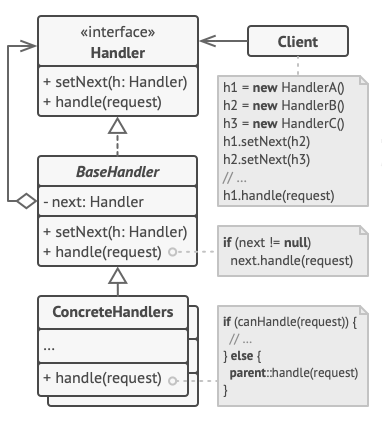
Chain of Responsibility là “a behavioral design pattern” cho phép bạn “pass request” dọc theo “chain of handlers”. Khi nhận được “a request”, mỗi “handler” quyết định xử lý hoặc chuyển tiếp tới “handler” tiếp theo “in the chain”.

### Solution

CoR dựa trên việc biến đổi “behaviors” riêng biệt vào bên trong “stand-alone objects” được gọi là “handlers” ( “behaviors” ở đây hiểu trong một tình huống cụ thể là các hành vi “checks” thông tin của request để xử lý một vấn đề nào đó). Trong trường hợp này, mỗi “check” sẽ được triết xuất tới “class” riêng của nó với “a single method” cái mà sẽ thực hiện việc “check”. “The request” cùng với “data” đi kèm với nó sẽ được “pass” tới “this method” dưới dạng “argument”.



### Structure



* Handler có 2 phương thức là setNext và handle. setNext đại là phương thức để thiết lập handler tiếp theo nếu handler này không xử lý và chuyển cho handler kế tiếp.
* BaseHandler – có chứa một field là –next, field này dùng để lưu lại địa chỉ của handler tiếp theo.

Xét theo kết cấu này ta có thể thấy một điều, đây là cấu trúc của một link-list khi mà mỗi handler là một node và node sau sẽ được tham chiếu bởi node trước qua các field.

## Command

Command là “behavioral design pattern” cái mà chuyển đổi “request” vào bên trong “a stand-alone object” “object” này chứa toàn bộ thông tin về “request”.

Vấn đề:

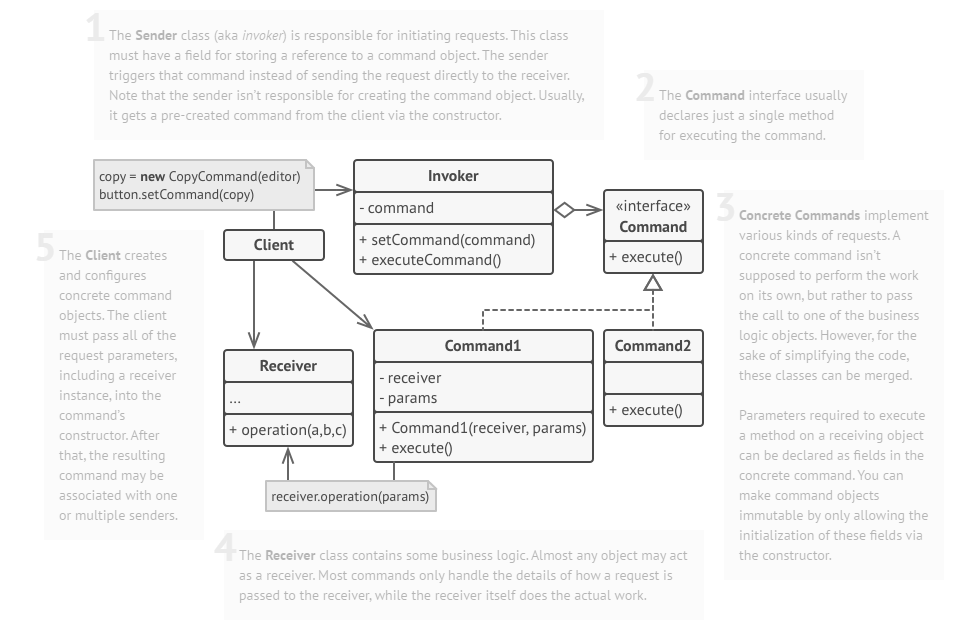
Chúng ta cần tạo một GUI, và trên GUI có nhiều button. Mỗi button giữ một chức năng khác nhau. Bình thường ta sẽ nghĩ đến một cách đó là tạo một abstract button và sau đó các specific button vd SaveButton, CancelButton, OpenButton.. sẽ thực thi abstract button đó.

Đồng ý là ta sẽ chi ra rất rõ ràng và cụ thể, nhưng ta sẽ có một hierarchy button khổng lồ khi mà sau này sẽ có thêm các button.

Ở đây nếu theo cách này là ta đang mô hình hóa button theo behavior của button, mỗi behavior tương đương với một button, button giờ không những phải take care UI, các thông số,.. mà nó còn phải take care cả behavior là làm gì. Như vậy là vi phạm nguyên tắc S trong SOLID và SOC (separate of concern) => một cách khác mà ta sẽ dùng đó là IOC – Inversion of control, chuyển đổi việc concern vấn đề beahvior làm gì sang một class riêng biệt.

Xét về mặt kết cấu ta thấy Command khá giống Abstract Factory khi dùng Bridge để switch giữa các thực thi phù hợp với điều kiện và hoàn cảnh trong runtime.

Với Command ta có một nhân tố khác biệt là Invoker, thằng này chính là người Invoke và cũng là người ghi lại lịch sử Invoke command để sau đó chúng ta có thể tiến hành Undo, Redo.

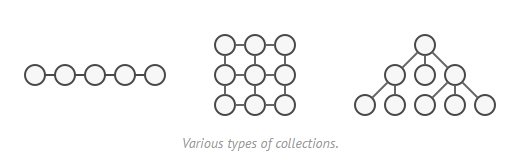


## Iterator

Là “a behavioral design pattern” cho phép bạn ghé thăm “elements of collection” mà không cần phơi thể hiện bên dưới của chúng (list, stack, tree,etc.)

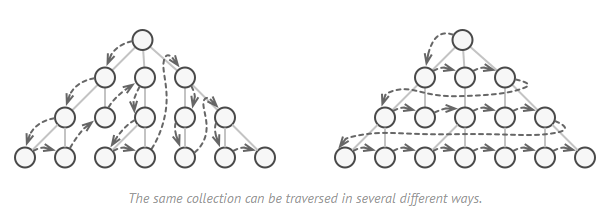
### Vấn đề

Collections là một trong những cái được sử dụng nhiều nhất trong chương trình. Tuy nhiên, một collection chỉ là một container cho “a group of objects”



Phần lớn “collections” lưu các phần tử của họ trong “simple lists”. Tuy nhiên, nhiều trong số chúng lại dựa trên stacks, trees, graphs hoặc các cấu trúc dữ liệu phức tạp khác.

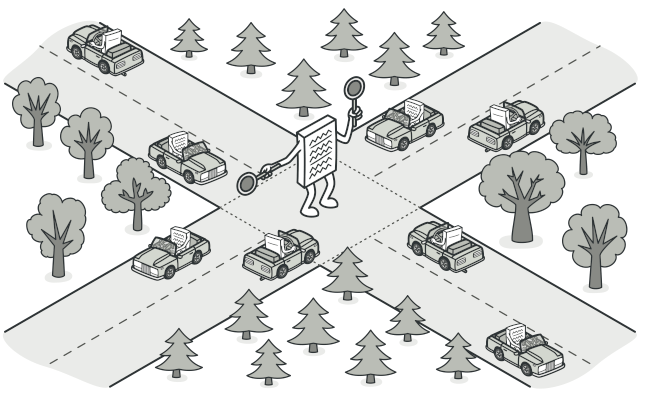
Làm thế nào để ghé qua một cách tuần tự các elements của cấu trúc dữ liệu phức tạp, như là tree.

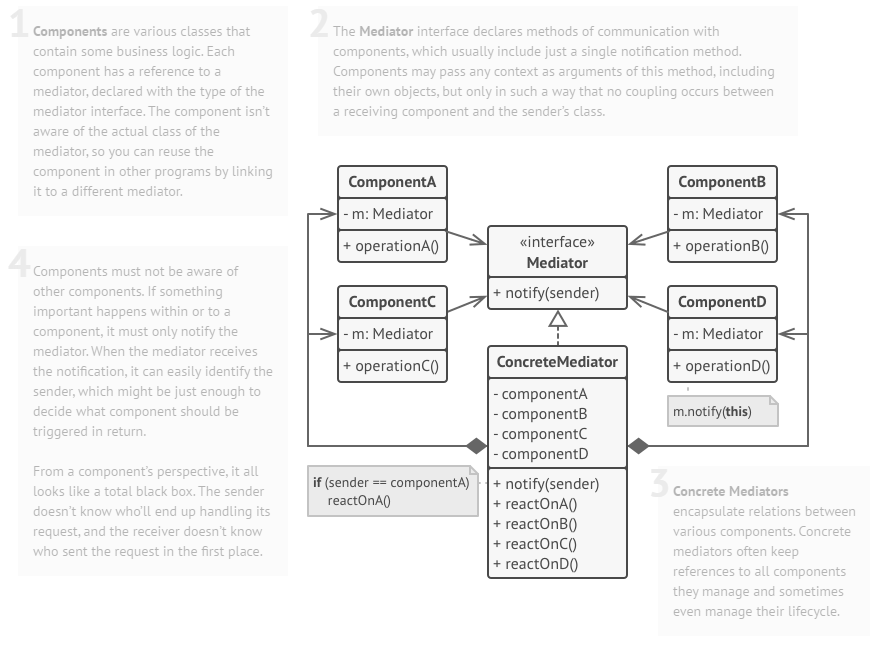


Iterator chưa phù hợp để tìm hiểu. Tìm hiểu lại sau nếu cần.

## Mediator

Là BDP cái mà cho phép bạn giảm sự hỗn loạn phục thuộc giữa các objects với nhau. The pattern hạn chế việc liên hệ trực tiếp của các objects và bắt chúng chỉ có thể làm việc với nhau thông qua mediator object.





## Memento

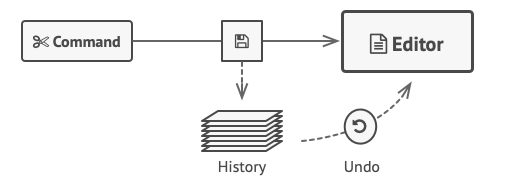
Là “behavioral design pattern” cho phép bạn lưu hoặc restore trạng thái trước của một object mà không cần quan tâm chi tiết việc thực thi của nó.

Nếu với Command chúng ta có thể Undo bằng cách lưu lại history của command và làm ngược lại, thì với Memento chúng ta sẽ không làm thế mà chúng ta lưu lại trạng thái object đó.

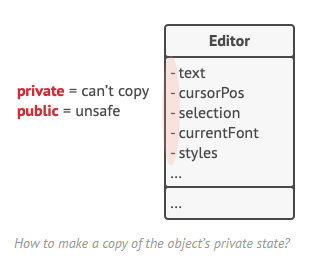
### Problem

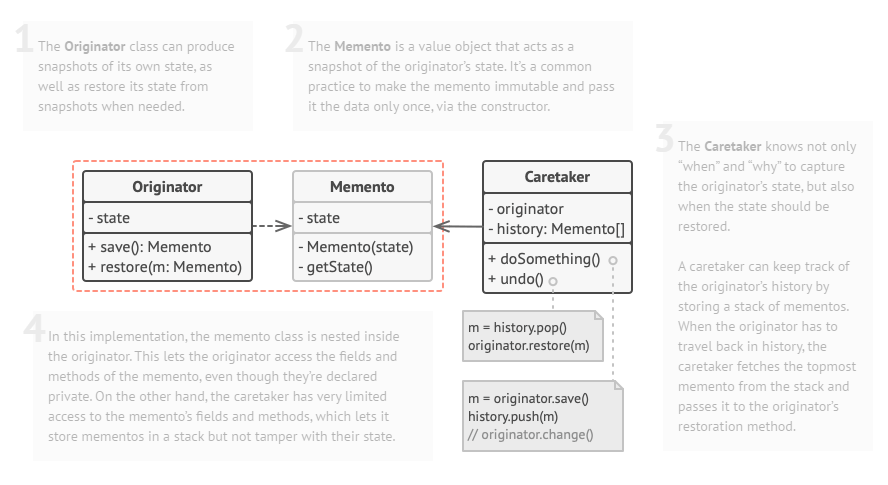
Tưởng tượng ta đang tạo một text editor app. Ban đầu chúng chỉ có chức năng như format text, insert,... nhưng sau này chúng ta muốn thêm một chức năng là Undo và Redo.

Để thực thi cái này, Trước khi thực hiện bất cứ hành động nào, app ghi chép trạng thái của toàn bộ objects và lưu nó vào trong một vài nơi lưu trữ. Sau đó khi mà một user quyết định revert a action, the app fetchs lastest snapshot từ history và sử dụng nó để restore the state của toàn bộ objects.



Như vậy, để ghi ra một bản sao state của object, có lẽ ta phải sao chép toàn bộ state của nó, nhưng nếu nó có các state ở private thì làm sao mà ta có thể copy được, vì nó đâu có phơi ra bên ngoài.





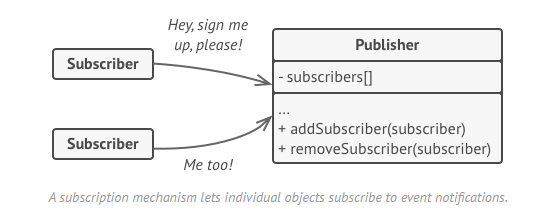
## Observer

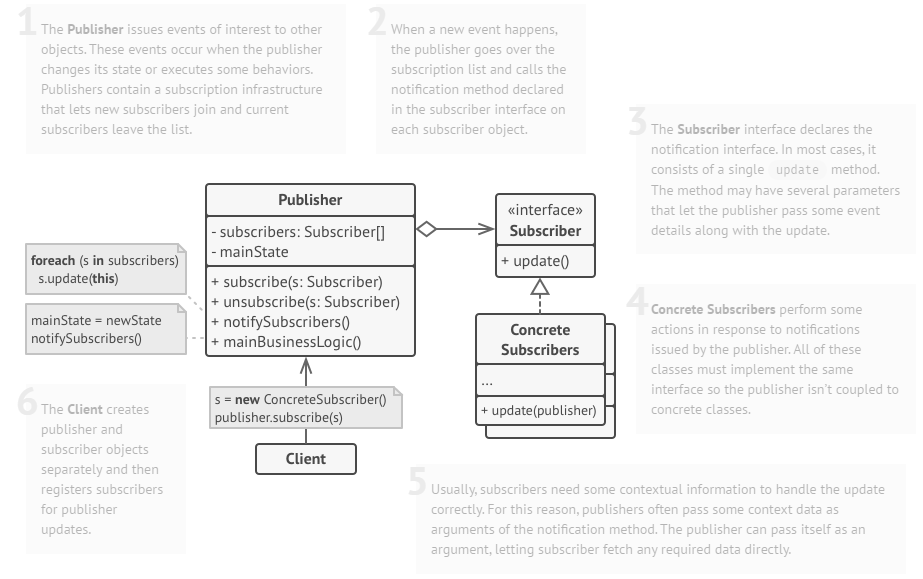
Là BDP chon phép định nghĩa một cơ chế đăng ký để thông báo tới multiple objects về bất kì events khi xảy ra sẽ được thông báo tới object observing.

Còn được gọi là : Event-Subcriber, Listener

Giả dụ ta có một Publisher với state thay đổi thì có những đối tượng quan tâm đến sự thay đổi đó của State thì gọi là Subcirbers.

The Observer pattern gợi ý rằng bạn thêm một cơ chế đăng ký tới publisher class để các objects cá nhân có thể subcribe hoặc unsubcribe từ luồng của events.





## State

Là BDP cho phép một object thay đổi behavior của chính nó khi state trong nó thay đổi.

