**Pro Javascript Design Patterns**

**Factory Pattern**

**When should the Factory Pattern be used**

* **Dynamic Implementations:** các class khác nhau có cùng interface
* Factory method: là hàm trả về kết quả phụ thuộc vào điều kiện để hàm khác dung mà ko cần quan tâm
* **Memoizing:** 1 hàm trong lúc thực thi sẽ ghi đè lên chính nó, vd hàm trả về XHR object, lúc đầu chạy thì sẽ xét môi trường để chọn XHR, sau đó sẽ tự ghi đè lên để lần sau chạy chỉ có XHR ban đầu chọn

Code ban đầu có thể tính toán phức tạp để lựa chọn code phù hợp, gọi là setup code

* Factory method là các interface nằm trong object? Dung để setup tất cả các object cần thiết
* Tóm lại là: dùng factory khi cần lựa chọn class ở thời điểm thực thi, gọi là encapsulate instantiation

**The Bridge Pattern**

* Bridge hữu ích cho việc thực hiện APIs, lập trình dựa vào sự kiện như JS
* Sử dụng nhiều cho call back của event listeners
* Khi dung hàm call back cho event listener, trong hàm có thể dung this.some, điều đó làm cho hàm ko độc lập.

Thay vì vậy, ta có thể tạo ra hàm đó và thêm số lượng tham số (this.some), hàm đó có thể dung ở chỗ khác

Sau đó ta dung 1 hàm trung gian (bridge) nhận this.some từ event và gọi hàm độc lập kia với tham số truyền vào

* Hàm accessor, mutator là các bridge vì nó là các hàm trung gian để truy xuất các biến private
* Dùng bridge để giảm sự phụ thuộc lẫn nhau
* Nên viết API trc khi dùng nó cho các mục đích khác nhau

**The Composite Pattern**

* Dùng mô hình này nếu ta có các object lồng vào nhau nhiều cấp và các hoạt động tác động lên các obj này
* Mô hình này sẽ truyền hoạt động từ object này đến object khác
* Trong các object lồng vào nhau chỉ có 2 loại: composite, leaf
  + Composite: chứa các composite # và leaf
* Khi thực hiện 1 tác vụ cho các object, ta chỉ cần gọi composite tổng
  + Vd: A.someMethod
* someMethod sẽ đc gọi tới khi gặp leaf và thực hiện ở leaf
* Dùng khi có nhiều chỗ cùng có 1 tác vụ (vd: validate form, ẩn hiện), số lượng chỗ ko cố định. Khi đó, ta tạo ra 2 loại object (composite, leaf), leaf chính là thành phần cần thực hiện tác vụ. Ta tạo ra các object cho các chỗ đó dựa vào 2 loại này, và quản lý bằng 1 mảng (mảng này nằm trong object composite)
* Cấu trúc object dùng để phân ra các chức năng thuộc các nhóm khác nhau

**The Facade Pattern**

* Đơn giản interface, giảm ràng buộc của các class với code sử dụng nó

**The Decorator Pattern**

* Cho phép thêm chức năng vào obj mà ko cần tạo ra subclass
* Giả sử có 1 số chức năng A, B, C, D. Có những lúc ta cần tạo 1 obj có chức năng của A và B, vậy ta cần tạo 1 class hoặc kế thừa từ 2 class có chức năng này, hoặc viết lại có chức năng tổng hợp của 2 chức năng này.

Dựa vào đó sẽ có 1 số tổ hợp các class đc tạo ra: A + B, A + B + C, A + D, B + C, …

* Dùng mô hình decorator để ko cần tạo ra các class tổng hợp đó, chỉ cần tạo ra các class cho từng chức năng
* Dễ bảo trì
* Decorator là obj bao obj #, thừa hưởng các chức năng của obj component, và thêm cái mới
* Decorator phải có các chức năng giống như component của nó, nghĩa là decorator và component có cùng interface
* Composite: chỉ truyền method cho các obj, tổ chức các obj, decorator: chỉnh sửa chức năng vào obj
* Các cách chỉnh sửa method của component
  + After method: Cùng là 1 method, khi nó đc decorator qua nhiều lớp sẽ thêm nhiều chức năng
  + Before method
  + Replace: dùng ở lớp cuối
* Function decorator: dùng bọc func #, dùng apply
* Khi decorator nhiều lần thì obj sẽ có đc nhiều đặc điểm, có cách nào đi ngược lại ko? Bỏ những đặc điểm khi ko còn cần thiết mà ko cần decorator lại từ đầu
* Decorator làm phức tạp cấu trúc code, làm dev sợ

**The Flyweight Pattern**

* Là 1 optimization pattern, dùng trong trường hợp nhiều object tương tự nhau đc tạo ra
* Cải thiện performance vì giảm bộ nhớ?
* Chuyển nhiều object thành 1 số object dùng chung
* Chia trạng thái của 1 object thành 2 loại: intrinsic data và extrinsic data
  + Intrinsic data: là thông tin cần cho các method của class
  + Extrinsic data: thông tin có thể đc loại bỏ khỏi class và lưu ở ngoài
* Lấy tất cả obj có cùng intrinsic state chuyển thành 1 obj
* Factory đc dùng để tạo ra obj dùng chung
* Chỉ tạo ra obj khi nó có intrinsic state khác với trc đó
* Manager object: lưu extrinsic state
* Khi gọi method của object, manager object sẽ truyền các extrinsic state như là các param
* 1 record = 1 object chung + data ngoài
* Manager chứa data, tham chiếu đến object của data đó, các method liên quan
* Giả sử có 1 class A với các property x, y, z, t, a, b, …. Nhưng bây giờ ta ko tạo ra 1 obj từ class như vậy, mà tách các pro ra làm 2 nhóm, 1 nhóm sẽ tạo ra obj.

Trở lại với mục tiêu để tạo obj ban đầu, nếu các obj đó có các property giống nhau thì có thể tách các property ra làm 1 obj dùng chung. Để giảm bộ nhớ, ta tách các property thành 2 nhóm, ví dụ: (x, y, z), (t, a, b); (x, y, z) gọi là intrinsic data, còn lại là extrinsic data. Tiếp theo ta chỉ tạo ra class có các property x, y, z, với các tổ hợp x, y, z khác nhau ta sẽ có các obj khác nhau. Bây giờ để tạo ra 2 obj có các pro (cả 2 nhóm), nếu pro nhóm 1 giống nhau cho cả 2 obj thì ta chỉ dùng 1 obj loại 1 cho cả 2

* Mục tiêu của flyweight là giảm lượng obj ko cần thiết về những obj dùng chung. Manager sẽ lưu, quyết định bao nhiêu object dùng chung đc tạo ra. Chuyển property thành các param truyền vào method

**The Proxy Pattern**

* Proxy là 1 obj điều khiển việc truy cập vào obj khác (A)
* Proxy có interface giống như obj A, A đc gọi là real object
* Proxy đc tạo trc real obj, và hoãn lại việc tạo obj A, chỉ tạo obj A khi thực sự cần, dùng khi obj A lớn và phức tạp. (virtual proxy)
* Proxy ko chỉnh sửa method như decorator, đơn giản interface như façade
* Khi gọi 1 method trên proxy, nó sẽ truyền cho real obj
* Remote proxy: nhận lệnh JS từ server, đưa vào eval để thực thi??? Dùng remote proxy để bọc lại.

**The Observer Pattern**

* Còn gọi là mô hình publisher-subscriber, dùng để quản lý các object, các hành động, trạng thái của nó
* Quan sát trạng thái của 1 object và biết khi nó thay đổi
* Trong mô hình này có 2 vai trò: người quan sát (observer – publisher) và người bị quan sát (observered – subscriber)
* Quan hệ giữa publisher – subscriber là many-many
* 2 phương thức để subscriber nhận data từ publisher: push và pull
* Subscriber có thể subcribe, unsubscribe, có thể lựa chọn nhận data qua chuyển phát hoặc tự đi lấy
* <http://www.dofactory.com/javascript/observer-design-pattern>
  + Định nghĩa sự phụ thuộc 1-n giữa các obj, để khi 1 obj thay đổi, tất cả các obj phụ thuộc sẽ biết và tự động cập nhật.
  + Các obj đăng ký 1 event và sẽ đc thông báo khi event xảy ra
* <https://en.wikipedia.org/wiki/Observer_pattern>
* Head first design patterns
  + Giúp cho những object biết khi có những thứ liên quan đến nó xảy ra
  + Subject và observers: phụ thuộc lỏng (loosely coupled)
  + Thiết kế “loosely coupled” giúp ta xây dung hệ thống OO linh hoạt để có thể xử lý những thay đổi