1. **The Strategy Pattern**: định nghĩa ra 1 family of algorithms (có thể là methods or các functions) và đóng gói chúng lại => để ta có thể hoán đổi chúng cho nhau trong lúc sử dụng tùy vào clients.
2. **Observer Pattern**: Sử dụng mqh 1-n, lấy 1 Subject làm nơi lưu trữ, chỉnh sửa data, và các Observers đăng kí với Subject đó sẽ đc update và notify thường xuyên về data đó (có thể sử dụng built-in Observable though not recommended) hoặc tự tạo mô hình riêng.
3. **Decorator pattern**: Khi muốn add thêm behaviours or functionailites cho 1 object dynamically aka chỉnh được trong runtime (chứ kp cho toàn class) có những “phụ kiện” đi kèm (như cốc coffee có toppings, player có items trên người) thì ta follow decorator pattern – build 1 object thông qua việc wrap các decorators ra bên ngoài layer by layer (hiện biết cách là thông qua constructor => wrap lại), và khi sử dụng tính toán thì sẽ được unbox đập hộp tính từ ngoài vào trong cùng.

Quan trọng:

* Decorators phải có cùng supertype với object mà chúng decorates.
* Decorators cũng có behaviours và attributes riêng.
* Decorators sẽ thay đổi behaviour components mà chúng wrap => nếu ta cần concrete component type (type object mà đc wrap) thì không nên dùng decorators.
* Should not overuse !!!

1. **Factory Method Pattern**: Defines 1 interface (or abstract class) for object creation aka calling constructors, và để subclasses – classes implement interface or abstract class đó decide việc instantiation. Pattern này giúp ta encapsulate instantiation of concrete types

Được sử dụng khi có nhiều instance của 1 super type cần đc instantiate mà ta không biết trong runtime phải khởi tạo loại nào !!!.

1. **Abstract Factory Pattern**: Provide 1 interface để tạo ra 1 families of dependent objects mà k cần specify concrete classes của chúng (Ví dụ ở trong eclipse Windows, trong package ducpad class PizzaStation)
2. **Singleton Pattern**: ensures a class has only one instance, and provides a global point of access to it. Ta khởi tạo 1 class có private constructor, và tạo 1 method return lại instance được khởi tạo trong class đó => sinh ra 1 instance only. Có 1 vài problems lquan đến singleton pattern:
3. Nếu khởi tạo trong hàm getInstance() thì dễ bị lỗi khi gặp multithreading (các steps bị xáo trộn dẫn tới tạo ra 2 instances của class).
4. Nếu khởi tạo luôn lúc đầu => instance sẽ sống đến khi program terminates
5. Nếu sử dụng từ khóa synchronized cho hàm getInstance() thì sẽ bị low performance

A screenshot of a social media post

Description automatically generated

1. **Command Pattern**:
   1. Encapsulate 1 request (**command**) thành 1 object – thành phần then chốt của pattern, từ đây giúp cho ta có thể sử dụng các requests như parameters => thay đổi tùy ý. Ví dụ như remote control, với 1 nút khi muốn bật đèn thì chuyển sang mode bật đèn (set command lightOn) thì khi ấy nếu bấm nút ta sẽ bật đèn. Khi chuyển sang mode bật tivi thì lại set mode bật tivi với cùng 1 nút bấm.
   2. Trong command sẽ có chứa **receiver (1 object mà ta muốn thực hiện hành động nào đó)** làm 1 instance variable và 1 hàm ví dụ như execute (). Hàm này sẽ được sử dụng để receiver gọi method của nó.
   3. Ta cần có 1 **invoker** – object để lưu các loại commands và tùy lúc gọi command cần thiết. Invoker chính là object ta đang cần design (ví dụ: remote control, keyboard ??). Ví dụ với remote control, ta sẽ có các methods buttonPressed() và bên trong sẽ gọi command.execute() cần thiết.
   4. Chú ý rằng ta cần có 1 command chung làm interface – program to an interface sau đó các concrete commands sẽ implements interface này.
   5. Applications của command pattern: cho queuing requests – xếp các requests vào queue để chờ được sử dụng (cho multithreading cases là chính). Ngoài ra còn sử dụng cho việc log requests, khi chạy command thí sẽ store vào stack chẳng hạn, khi app sập thì có thể lấy lại được
2. **Adapter Pattern**: converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces. Đại khái là: tạo 1 class Adapter implements interface của bên Client, nhưng bên trong sử dụng reference variable của 1 interface khác, và khi implement methods của interface bên client thì sử dụng methods của reference variable kia => convert.

A screenshot of a cell phone

Description automatically generated

1. **Façade pattern**: provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Đại khái là những class lây nhây sẽ được encapsulate vào 1 class, và những methods của chúng sẽ đc đem ra sử dụng để client chỉ cần gọi façade interface thôi k cần phải gọi nhiều.
2. **Template Method Pattern**: defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure. Đại khái là coi các algorithms là các methods, có 1 method gọi những methods này, nhưng thay vì implement hết, thì để lại 1 số methods abstract, để cho subclasses khi extends sẽ implements => thay đổi redefine certain steps trong algo mà k thay đổi structure của toàn method. Ngoài ra, 1 loại method hook() cũng được sử dụng trong pattern này, nhằm mục đích nếu subclass muốn bổ sung algo gì có thể override lại để thêm thắt.

Na ná Strategy pattern ở chỗ cả 2 đều group methods / algorithms lại, nhưng strategy encapsulate vào 1 class và sử dụng, còn đây template group lại và để abstract, cho subclasses thay đổi.

1. **The Iterator Pattern**: provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Đại khái là sử dụng iterator để access tới các instance variables, cụ thể là các collections. Thông qua iterator, thì dù object có lưu elements dưới dạng gì đi chăng nữa, thì cũng quy về 1 mối là iterator => thống nhất. Ngoài ra cũng giúp client không cần quan tâm tới object stores data kiểu gì, ArrayList hay Vector.

Note: Đối với các collections thì không cần implements Iterator, tuy nhiên nếu sử dụng loại data structure khác thì cần implement, theo mô hình như ảnh dưới:

A screenshot of a cell phone

Description automatically generated

1. **Composite Pattern**: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Đại khái là sử dụng chung 1 interface cho toàn bộ các components liên quan (composite and individual objects), và những components này được lưu dưới dạng **tree**. Nên nhớ là 1 object có thể là composite (parent node) có thể là individual (leaf node). Khi client sử dụng, thì chỉ quan tâm tới interface tổng, chứ không cần check xem objects đang gọi thuộc type gì nữa. Cái này same same với Decorator Pattern, nhưng mục đích khác nhau.

**NOTE**:

* Composite pattern kết hợp với iterator rất tốt thông qua class CompositeIterator và NullIterator tự tạo implements Iterator. Phần implementation tham khảo trong code DesignPatterns trên git.
* Tradeoff của Composite Pattern: vì lưu dưới dạng tree nên khoản delete add sẽ rất lằng nhằng. Sắp xếp theo order cũng lằng nhằng