Tham khảo từ: <https://tienthanh214.github.io/design%20pattern/design-pattern/>

1. **Strategy**

aka: Policy

Đóng gói một họ các thuật toán lại vào các class, các class này có chung interface. Nhờ tính đa hình, có thể dễ dàng thay đổi giữa các thuật toán kể cả trong run-time (bằng cách gán object interface bằng instance của concrete class nào đó cùng họ). Giúp việc mở rộng, phát triển các thuật toán mới không ảnh hưởng đến client sử dụng nó, giảm sự phụ thuộc lẫn nhau.

**Problem:**

Giả sử mình muốn viết con game nhập vai, trong game có nhiều loại vũ khí (súng, kiếm gỗ, thần kiếm, đũa phép thuật, blabla…). Và tất nhiên nhân vật có thể sử dụng vũ khí nào tùy ý. Trong quá trình chơi, nhân vật cũng có thể thay đổi vũ khí.

Tất nhiên là trong tương lai (nếu game thành công :3) thì mình sẽ tạo thêm đa dạng vũ khí hơn hoặc nâng cấp cho các vũ khí cũ. Rõ ràng là mình sẽ muốn làm cho sự thay đổi về vũ khí này trở nên dễ dàng và không ảnh hưởng nhiều đến Nhân vật.

**Solution:**

Đóng gói mỗi loại vũ khí trong một class riêng (Gun, Sword, MagicWand, …), chúng có chung interface là Weapon. Trong class Character mình sẽ lưu một object weapon có kiểu là Weapon để reference đến vũ khí nhân vật dùng.

Bằng kiến thức cơ bản về polymorphism, object weapon này có thể sẽ là bất kì loại vũ khí gì miễn nó là subclass của Weapon, và vì nó là biến nên có thể thay đổi bất kì lúc nào.

Note

Có thể sử dụng anonymous class để tạo ra object thay vì phải tạo ra thêm class.

Nhưng Dart không support anonymous class

1. Decorator

Decorator cho phép thêm hành vi bằng cách đặt object vào trong 1 object khác, "object khác" đó gọi là decorator.

<https://www.youtube.com/watch?v=to5yJ54AxOg>

**Quay lại với Design pattern,** design pattern được sử dụng để solve một cái problem, **nếu bạn không có problem thì bạn không cần dùng Design Pattern.**

Trong description bạn có ghi: Chúng ta có thể sử dụng Decorator để tách những phần nhỏ ra và thêm vào nếu muốn trong runtime.

=> Nếu bạn có thể tách object ra những phần nhỏ hơn thì nghĩa là bạn có khả năng thay đổi object lúc compile mà không cần làm trong runtime.

Decorator nên ứng dụng khi bạn không thể (hoặc không muốn) thay đổi object nhưng vẫn muốn thêm hành vi.

**Ví dụ:**

Bạn có class MilkTea với method Build() để làm 1 ly trà sữa trân châu bình thường. Nếu bạn muốn thêm Pudding mà không phải sửa class MilkTea thì làm thế nào.

Bạn tạo một class BaseMilkTeaDecorator kế thừa MilkTea với constructor có parameter là MilkTea và attach behaviour vào trong Build{} method

class BaseMilkTeaDecorator{

MilkTea wrappee;

BaseMilkTeaDecorator(MilkTea wrappee){}

void Build(){}

}

Và mỗi loại topping thì bạn kế thừa từ cái BaseDecorator.

Client nếu muốn đặt trà sữa trân trân châu chocolate pudding thì có thể làm như sau:

var milkTea = new PuddingMilkTea(new ChocolateMilkTea(new MilkTea()));

milkTea.Build();

Tham khảo từ 1 nguồn khác:

<https://refactoring.guru/design-patterns/decorator>

* **Problem**

Giả sử cần viết 1 class để gửi noti đến người dùng qua email. Nhưng sau này cần thông báo qua sms, facebook nữa. Mà ta không muốn sửa class để gửi email cũ.

* Thì ta có thể dùng decorator pattern

Sự khác nhau giữa việc dùng kế thừa và dùng decorator pattern là gì?

VD: Ban đầu ta có 1 class Notifier để chỉ để gửi email đến những Listener

Bây giờ muốn gửi email rồi gửi luôn qua sms, rồi lại gửi tiếp đến facebook thì phải làm sao?

Có 2 cách để giải quyết:

+ Dùng kế thừa: tạo class SMSNotifier kế thừa Notifier rồi override lại function send() rồi trong hàm override này gọi đến super.send() và triển khai tiếp cái việc gửi sms. Còn để gửi tin qua fb thì làm tương tự FbNotifier lại kế thừa Notifier,… Nhưng giờ muốn cả gửi email, gửi sms, gửi qua FB thì sao. => FbNotifier cần kế thừa SMSNotifier.

Với 3 cách gửi tin đã thấy phức tạp rồi, giờ cần gửi qua thư, qua zalo,… thì nếu chỉ cần gửi qua email, sms, zalo mà ko cần qua FB, qua thư thì việc kế thừa sẽ thật là ối giời ơi. Việc kế thừa sẽ là “tổ hợp” của các option, càng nhiều option thì càng cần kế thừa nhiều.

Hơn nữa kế thừa bị fix cứng cở compile time. Ta phải quy định những class kế thừa nhau từ lúc code rồi.

+ Dùng decorator pattern: Thích dùng option nào thì enable nó lên là được.

*void* demoDecorator({

*required* String message,

*required* bool smsEnabled,

*required* bool facebookEnabled,

}) {

  Notifier stack = EmailNotifier();

  if (smsEnabled) {

    stack = SMSNotifier(stack);

  }

  if (facebookEnabled) {

    stack = FacebookNotifier(stack);

  }

  stack.send(message);

}

1. Factory

* Factory method design pattern

**Mục đính**: Định nghĩa một interface để khởi tạo object, và để class con quyết định cái class nào sẽ được khởi tạo. Giao việc khởi tạo một đối tượng cụ thể cho lớp con. (Trong ví dụ sẽ là class cha Application và class con MyApplication)

enum DocumentType {

  BLANK,

  REPORT,

  CERTIFICATE,

}

abstract class Application {

  Document createDocument(DocumentType type);

  Document newDocument(DocumentType type) {

    Document document = createDocument(type);

    document.load();

    document.show();

    return document;

  }

}

Ví dụ ta có một abstract class Application và có 1 abstract function là createDocument. Class nào kế thừa class Application sẽ đảm bảo nhiệm vụ override lại hàm này để return ra kiểu document phù hợp.

class MyApplication extends Application {

  @override

  Document createDocument(DocumentType type) {

    switch (type) {

      case DocumentType.BLANK:

        return BlankDocument();

      case DocumentType.REPORT:

        return ReportDocument();

      case DocumentType.CERTIFICATE:

        return CertificateDocument();

    }

  }

}

*void* demoFactoryMethod2() {

*final* Application application = MyApplication();

*// Giả sử user sẽ chọn những loại document nào đó ở runtime*

  DocumentType userChoice = DocumentType.CERTIFICATE;

*// Thì đối tượng được tạo ra sẽ tương ứng với lựa chọn của user*

  application.newDocument(userChoice);

}

* Abstract factory