Phần 1 - Tổng quan

DESIGN PATTERNS FOR BEGINNERS

Biên tập bởi
Đội ngũ CodeGym

CODEGYM
Raising the bar

DESIGN PATTERN FOR BEGINNER

PHẦN 1:

TỔNG QUAN VỀ DESIGN PATTERN

CODEGYM
Raising the bar

ĐỘI NGŨ BIÊN TẬP

Nguyễn Khắc Nhật

Nguyễn Bình Sơn

Nguyễn Khánh Tùng

Phan Văn Luân

CODEGYM
Raising the bar

LỜI TỰA

Làm trong ngành công nghiệp phát triển phần mềm, bạn đã từng gặp phải trường hợp như thế này?

Bạn làm qua nhiều dự án khác nhau và nhận ra rằng, trong những dự án đó, mình luôn dùng một phương pháp, cách làm theo một khuôn mẫu nào đó để giải quyết các vấn đề gặp phải, khi chúng tương đồng với nhau và đây là những thứ thường xuyên. Lặp lại, lặp lại…

Nếu có, đó chính là lúc mà bạn cần đến Design Pattern!

Được xây dựng theo dạng "template" - Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết trong tình huống cụ thể.

Tiếp theo đây, bạn có nghĩ mình phù hợp để đọc cuốn eBook này?

Điều quan trọng tôi muốn nói rằng: Design Pattern không dành cho những bạn mới bắt đầu tìm hiểu về lập trình. Muốn tìm hiểu và học được Design Pattern, bạn cần nắm cơ bản được kiến thức OOP đặc biệt là về abstract class, interface và static.

Không dành cho người mới tìm hiểu về lập trình, vậy tại sao tựa sách lại là *"for Beginners"*. Ở đây, chúng tôi muốn đem những người mới, những kẻ "dummy" đến với Design Pattern. Họ sẽ là những người bắt đầu làm quen với các "mẫu" và áp dụng nó để phát triển hơn kỹ năng, tay nghề của mình!

--

Những nội dung dưới đây là kiến thức được tổng hợp và biên tập lại từ đội ngũ sản xuất. Bằng tất cả sự nỗ lực để đem tới những kiến thức thực sự cần thiết và trọng tâm nhất cho những người bắt đầu tìm hiểu về Design Pattern.

Trong quá trình biên tập, đôi khi không tránh khỏi những sai sót, rất mong nhận được sự đóng góp của các anh, chị, em để cuốn eBook được hoàn thiện hơn.

Thân mến,

Đội ngũ biên tập!

CODEGYM
Raising the bar

MỤC LỤC

Bài 1: Design pattern là gì? 1
Protected: Design Pattern là gì? 1
Design Principle 3
Bài 2. Design Pattern là gì và những điều không thể bỏ qua 9
Design Pattern là gì? 9
Điều gì khiến Design Pattern trở nên quan trọng? 10
Để học được Design Pattern, bạn cần gì? 11
Các loại Design Pattern 11
Bạn có hứng thú với Design Pattern? 13
Bài 3. Nhập môn Design Pattern theo phong cách kiếm hiệp 13
Nhập đề 13
Hỏi thế gian DS là chi, mà bọn Dev thề nguyền sống chết 14
Design Pattern Kiếm Phổ 16
Khẩu quyết nhập môn Design Pattern 16
Thay lời kết 18
Bài 4: Design Pattern sự tiến hóa trong lập trình 18
1. Singleton Pattern 20
2. Factory Pattern 22
3. Composite pattern 26
4. Strategy pattern 31
5. Model - View - Controller MVC 36
6. Lời kết 42
Bài 5. Design Patterns: Best Practices for Application Development 42
23 Gang of Four Design Patterns 44
Bài 6. Vỡ lòng về bộ nguyên tắc thiết kế SOLID 45
Kiến trúc 45
Nguyên tắc thiết kế 46
SOLID 46
Nguyên tắc Đơn Trách Nhiệm 47
Nguyên Tắc Đóng/Mở 51
Nguyên Tắc Thay Thế Liskov 55

Nguyên Tắc Phân Tách Giao Diện 59
Nguyên Tắc Đảo Ngược Phụ Thuộc 61
Kết luận 62
Bài 7. Sự khác biệt giữa thiết kế tốt và thiết kế tồi trong kỹ thuật phần mềm 63
Bài 8. Giải thích mô hình MVC thông qua ... cốc trà đá 64
Nếu bạn từng đi uống trà đá, thì bạn đã hiểu được MVC rồi 65
Mô hình MVC là gì? 65
Bài học rút ra là gì? 67
Quay trở lại vấn đề lập trình web 68
Tổng kết 68
Bài 9. Sự khác biệt giữa các mẫu thiết kế MVC và MVT 68
Mẫu thiết kế MVC (Model-View-Controller) 68
Mẫu thiết kế MVT (Model-View-Template): 69
Bài 10. Kiến trúc lục giác trong Java 70
Bài 11. Những Design Pattern thường dùng trong Android 78
Lời mở đầu 78
Phân loại design pattern 78
Structural Patterns 81
Behavioral Patterns 83
Kết luận 84

CODEGYM
Raising the bar

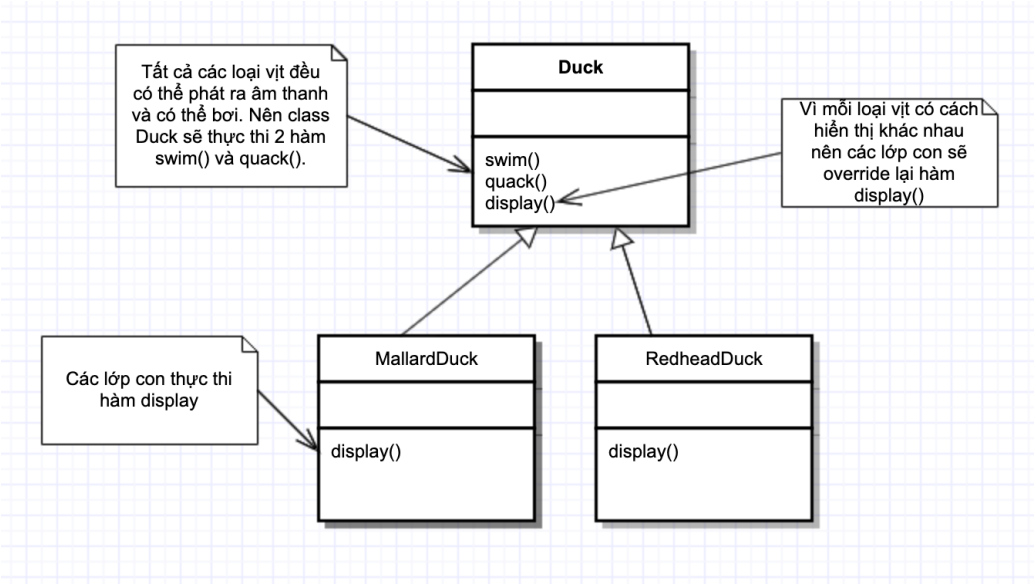
Bài 1: Design pattern là gì?

Protected: Design Pattern là gì?

Bắt đầu với ứng dụng mô phỏng Duck đơn giản:

Nam làm việc cho công ty mô phỏng về game có tên là *SimUDuck*. Game thể hiện nhiều trạng thái khác nhau của *vịt* về hành vi *bơi* và tiếng *kêu*.

Thiết kế ban đầu sử dụng hướng đối tượng (OO) bằng cách tạo 1 class Duck làm class cha để cho các lớp con thừa kế.

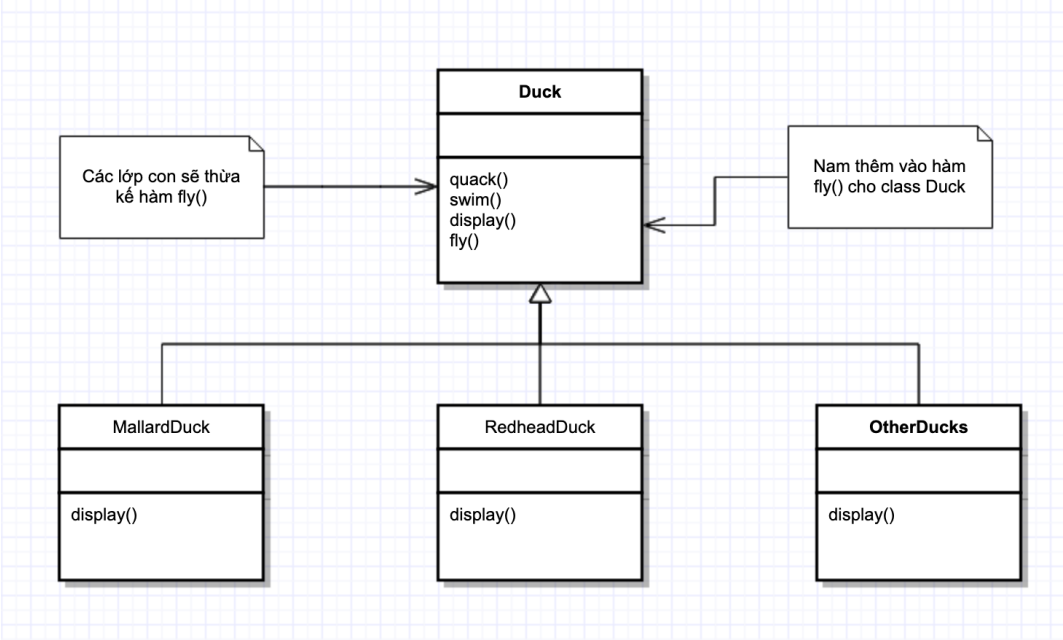


Vào cuối năm, do áp lực gay gắt với các đối thủ cạnh tranh. Sau 1 tuần suy xét cẩn trọng giám đốc quyết định phải tạo ra 1 bước đột phá mới.

Các chú vịt cần phải biết bay

Những chú *vịt biết bay* là chìa khoá của đột phá để giúp đánh bại các đối thủ cạnh tranh *vịt biết bơi*. Và đương nhiên, quản lí của Nam OK vấn đề này, và nói sẽ làm trong 1 tuần là xong ngay.

Và đây là công việc của Nam: Mình chỉ cần thêm method fly() vào lớp Duck (parent class) cho tất cả các lớp con thừa kế là xong ngay!



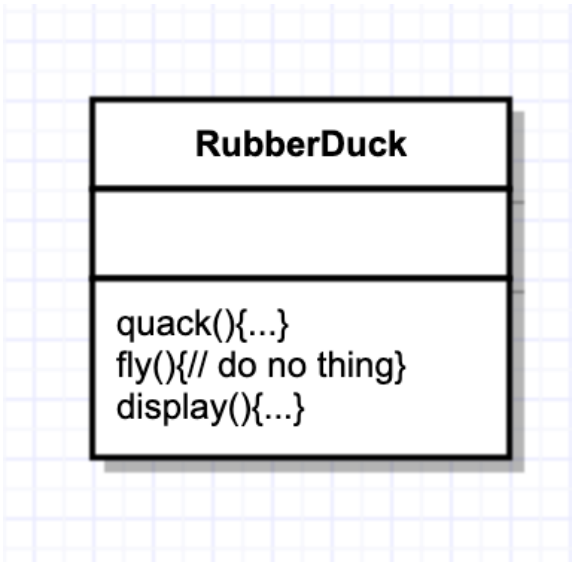
Nhưng có vấn đề xảy ra:

Giám đốc điện thoại cho Nam: "Cậu đùa với tôi ah, tôi đang chạy demo của cậu: Những chú *vịt cao su* làm sao biết bay!"

Nam quên 1 điều là những chú *vịt cao su* không thể bay. Và anh ấy nhận ra 1 điều rằng, kế thừa cũng không giúp gì nhiều trong việc tái sử dụng, bảo trì code.

Cùng xem xét lại về kế thừa

Vì *vịt cao su* có tiếng *kêu* khác với vịt thường, và vịt cao su không thể *bay* được. Chúng ta phải *Override* lại hàm quack() (kêu) và hàm fly() (bay) của *vịt cao su*.



Nhưng còn vịt gỗ (WoodenDuck) thì sao chúng không thể *kêu* hoặc *bay* được?

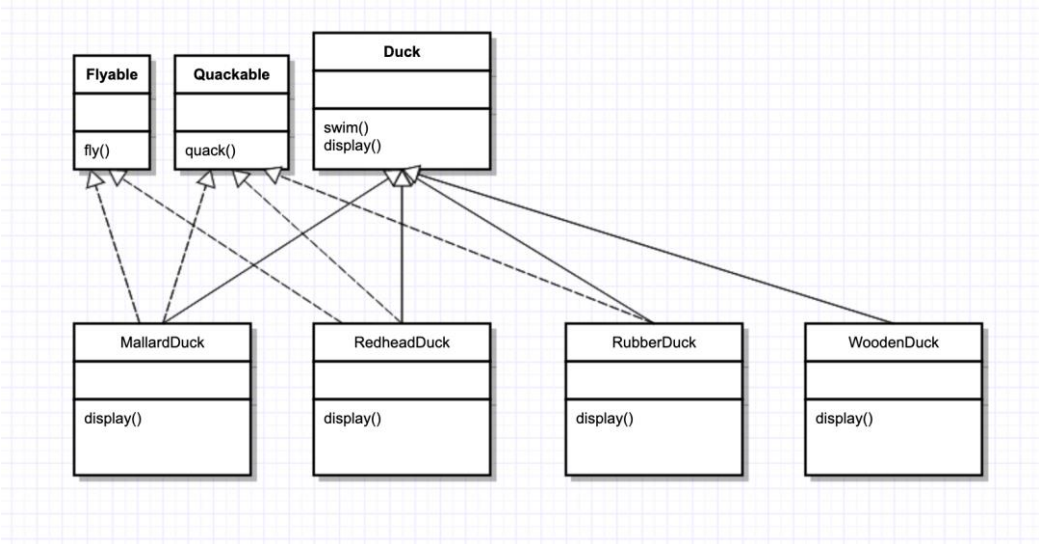
Interface thì như thế nào?

Nam nhận ra rằng, kế thừa sẽ làm mã của anh ta lặp lại (duplicate code), và không thích hợp cho những mã bị thay đổi thường xuyên.

Trong đó hàm fly() và quack() sẽ bị ảnh hưởng bởi yêu cầu mới.

Nam quyết định đập lại mã của mình bằng cách: đem hàm fly() và quack() ra khỏi class Duck và tạo interface Flyable với hàm fly() và interface Quackable với hàm quack().

Chỉ những chú vịt có thể bay mới implement interface Flyable, và những chú vịt có thể phát ra tiếng kêu thì sẽ implement interface Quackable.



Nhưng giải quyết bằng cách này kết quả còn tồi tệ hơn nữa.

Theo cách kế thừa: bạn chỉ cần override lại vài hàm.

Theo cách interface: bạn phải override lại tất cả những chú vịt có thể bay và có thể phát ra tiếng kêu, hiện tại ta có 70 chú vịt như thế…

Bạn sẽ làm thế nào nếu bạn là Nam?

Chúng ta biết rằng không phải tất cả các lớp con sẽ kế thừa hàm fly() và quack(), vì thế kế thừa không phải là đáp án đúng.

Còn với interface, sẽ có nhiều class implement lại hàm fly, quack() của interface Flyable và Quackable, điều này sẽ hạn chế lại việc tái sử dụng code.

Và đó là tại sao chúng ta cần *Design Pattern*.

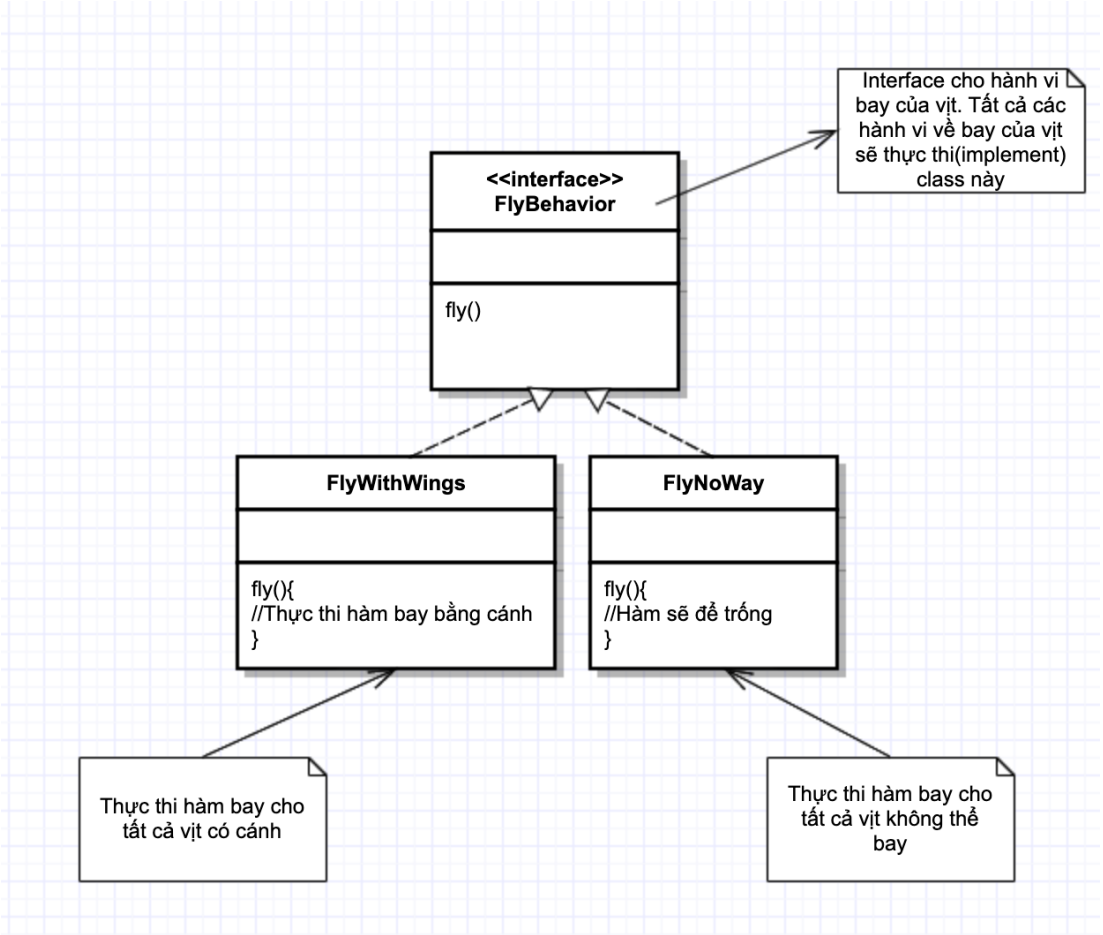
Design Principle

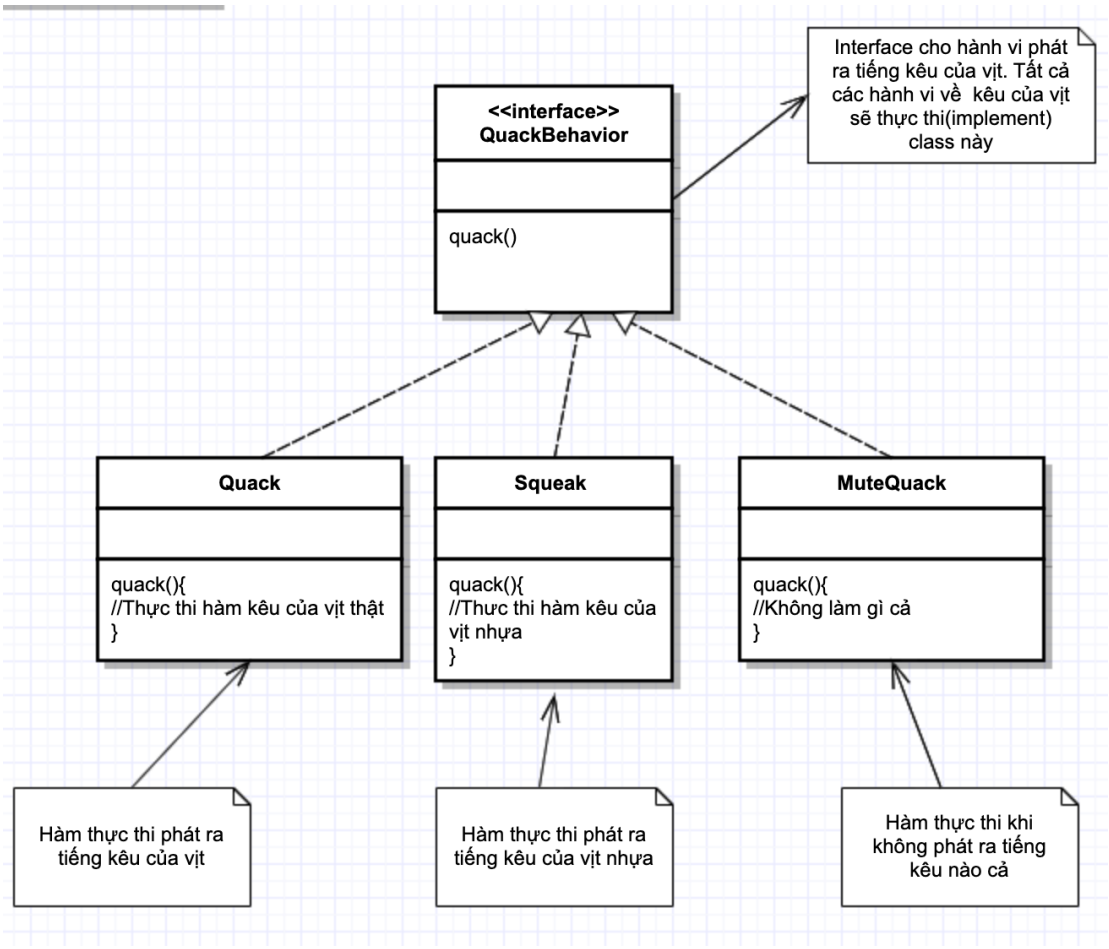
Identify the aspects of your application that vary and separate them from what stays the same.

Tạm dịch là: Tìm ra những phần hay thay đổi trong ứng dụng và đóng gói chúng, để không ảnh hưởng tới phần chung của hệ thống.

Thiết kế hành vi (behavior) cho những chú vịt

Nam nhận ra rằng, hàm `fly()` và `quack()` thay đổi thường xuyên nên sẽ được đóng gói riêng biệt.



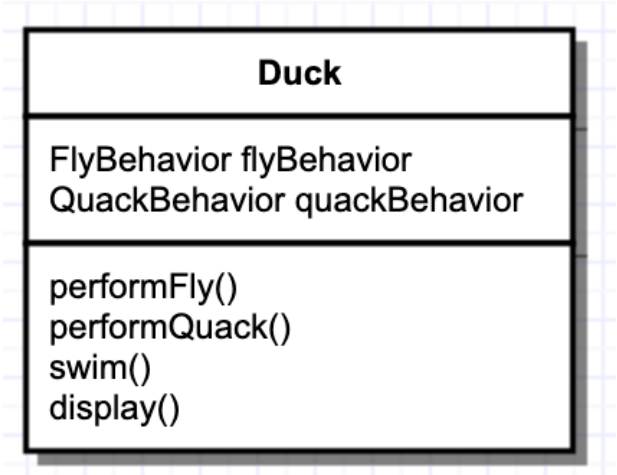


Với thiết kế này, các đối tượng có thể tái sử dụng lại hàm fly và quack.

Chúng ta có thể thêm mới hành vi mà không gây ảnh hưởng gì đến các đối tượng hiện có.

Tích hợp hành vi cho class Duck

Thay vì gọi hàm `fly()` và `quack()` trong lớp Duck. Nam sẽ khai báo `FlyBehavior` và `QuackBehavior` như sau:



Nam thêm 2 thuộc tính vào lớp Duck là flyBehavior và quackBehavior. Kiểu của chúng là interface.

Với flyBehavior và quackBehavior ta có thể thay đổi cách bay hoặc cách phát ra tiếng kêu trong thời gian thực thi (runtime).

Hàm thực thi của Duck

Với hàm thực thi này. Ta không cần quan tâm tới loại thực thi là gì. Thứ mà chúng ta quan tâm là chúng thực thi được được đúng chức năng hay không.

```
public class Duck {
    QuackBehavior   quackBehavior;
    FlyBehavior     flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }

    public void performFly() {
        flyBehavior.fly();
    }
}
```

Hàm thực thi cho lớp con

```
public class MallardDuck extends Duck{
    public MallardDuck() {
        flyBehavior = new FlyWithWings();
        quackBehavior = new Quack();
    }

    public void display(){
        System.out.println("I'm real MallardDuck");
    }
}
```

Vì flyBehavior và quackBehavior là interface, nên trong constructor MallardDuck phải khởi tạo đối tượng thực thi cho flyBehavior và quackBehavior.

Và MallarldDuck là vịt thật, nên có thể bay được, và phát ra tiếng kêu thật.

Code tổng hợp

```
public interface QuackBehavior {
    void quack();
}
public interface FlyBehavior {
    void fly();
}
public abstract class Duck {
```

```
    QuackBehavior   quackBehavior;
    FlyBehavior     flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }

    public void performFly() {
        flyBehavior.fly();
    }

    public abstract void display();
    public void swim(){
        System.out.println("Tất cả vịt đều có thể bơi, bao gồm vịt
");
    }
}
public class FlyWithWings implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("Bay bằng cánh");
    }
}
public class FlyNoWay implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("Tôi không thể bayy");
    }
}
public class Quack implements QuackBehavior{
    @Override
    public void quack() {
        System.out.println("Perform quack!");
    }
}
public class Squeak implements QuackBehavior{
    @Override
    public void quack() {
        System.out.println("Tôi là vịt nhựa!");
    }
}
public class MuteQuack implements QuackBehavior{
    @Override
    public void quack() {
        System.out.println("Tôi.... không thể nói!");
    }
}
public class MiniDuckSimulator {
    public static void main(String args[]) {
        Duck mallardDuck = new MallardDuck();
        mallardDuck.performQuack();
        mallardDuck.performFly();
```

```
    }
}
```

Kết quả hiển thị

```
Tôi là vịt thật!
Bay bằng cánh
```

Thay đổi hành vi (behavior) trong thời gian chạy (runtime)

Ta thêm 2 hàm sau vào lớp Duck

```
public abstract class Duck {
    .....
    public void setFlyBehavior(FlyBehavior flyBehavior) {
        this.flyBehavior = flyBehavior;
    }

    public void setQuackBehavior(QuackBehavior quackBehavior) {
        this.quackBehavior = quackBehavior;
    }
}
```

Tạo thêm 1 lớp *vịt* mới

```
public class ModelDuck extends Duck{
    public ModelDuck() {
        flyBehavior   = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display(){
        System.out.println("Tôi là vịt mẫu!");
    }
}
```

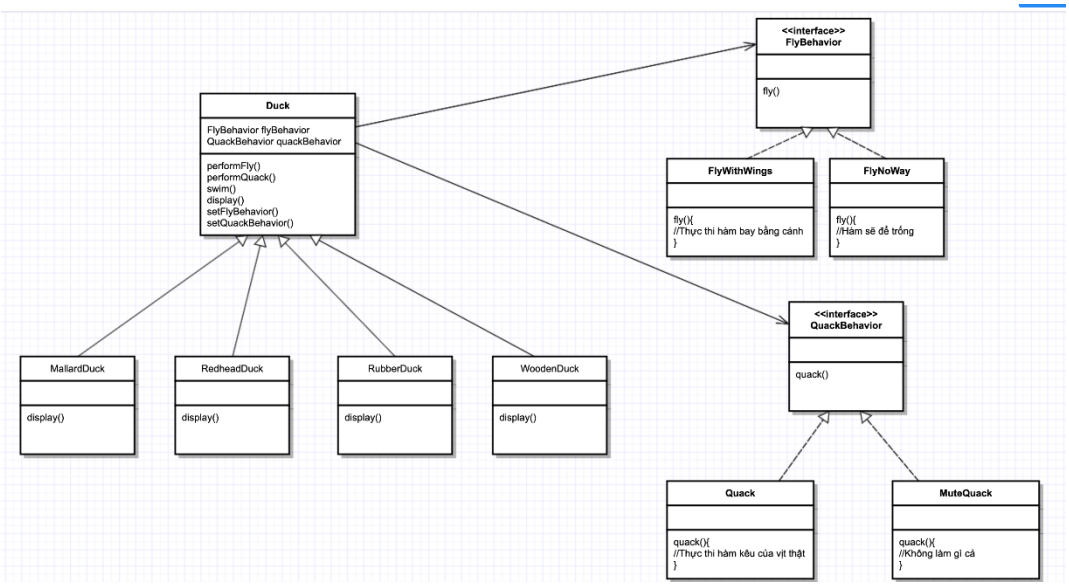
Tạo thêm 1 hành vi mới cho cách bay

```
public class FlyRocketPowered implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("Tôi bay nhanh như tên lửa!");
    }
}
public class MiniDuckSimulator {
    public static void main(String args[]) {

        Duck modelDuck = new ModelDuck();
        modelDuck.display();
        modelDuck.performFly();
        //Thay đổi cách bay thành tên lửa cho ModelDuck
        modelDuck.setFlyBehavior(new FlyRocketPowered());
        modelDuck.performFly();
    }
}
```

Tôi là vịt mẫu!
Tôi không thể bay
Tôi bay nhanh như tên lửa!

Như bạn đã thấy sau khi gọi hàm setFlyBehavior thì cách bay của modelDuck sẽ thay đổi.

Bức tranh tổng thể về Strategy Design Pattern:



Bài 2. Design Pattern là gì và những điều không thể bỏ qua

Các chuyên gia phần mềm có thể đã quá quen thuộc với thuật ngữ **"Design Pattern – Mẫu thiết kế"** (sau đây viết tắt là DP), nhưng thực tế cũng còn khá nhiều nhiều người không biết Design Pattern là gì. Do đó, người ta thường không thấy được các DP có giá trị như thế nào và lợi ích nó mang lại cho quá trình phát triển phần mềm, đặc biệt là trong các lĩnh vực bảo trì và tái sử dụng mã ra sao. Hãy cùng tóm tắt các tính năng nổi bật của một DP điển hình và đi đến một định nghĩa để bạn sẽ biết DP là gì và bạn có thể mong đợi những gì khi kết hợp DP vào thiết kế của bạn.

Design Pattern là gì?

Design Pattern (DP) được định nghĩa là các giải pháp đã được thử nghiệm để giúp giải quyết các vấn đề về thiết kế.

Các DP có nguồn gốc từ Christopher Alexander, một kỹ sư xây dựng đã đúc rút qua kinh nghiệm trong việc giải các vấn đề thiết kế vì chúng liên quan đến các tòa nhà và thị trấn. Alexander nhận ra rằng các cấu trúc thiết kế nhất định, khi được sử dụng hết lần này đến lần khác đều sẽ dẫn đến hiệu quả mong muốn. Ông đã ghi lại và công bố rộng rãi về kinh nghiệm này để những người khác có thể hưởng lợi.

Khoảng năm 1994, các chuyên gia phần mềm đầu tiên đã bắt đầu kết hợp các nguyên tắc của Alexandre vào việc tạo ra tài liệu mẫu thiết kế ban đầu như một hướng dẫn cho các nhà phát triển mới. Sự việc này sau đó được lan rộng và đạt đến đỉnh cao, đưa ra 23 mẫu dựa trên kinh nghiệm của các tác giả tại thời điểm đó. Những mẫu này được chọn vì chúng đại diện cho các giải pháp cho các vấn đề phổ biến trong phát triển phần mềm.

Tóm lại, Design Pattern là một kỹ thuật trong lập trình hướng đối tượng, được các nhà nghiên cứu đúc kết và tạo ra các mẫu thiết kế chuẩn. Và DP không phải là một ngôn ngữ lập trình cụ thể nào cả, nó có thể sử dụng được trong hầu hết các ngôn lập trình có hỗ trợ OOP hiện nay.

Điều gì khiến Design Pattern trở nên quan trọng?

Như đã nói, DP đang được ứng dụng trong hầu hết các ngôn ngữ lập trình có hỗ trợ OOP hiện nay. Tại sao nó lại phổ biến đến vậy?

Design Pattern giúp ích trong "giao tiếp", học tập và có cái nhìn sâu sắc hơn

- Trong thập kỷ qua, các mẫu thiết kế đã trở thành một phần không thể thiếu trong kho kiến thức của các nhà phát triển. Điều này thực sự giúp ích trong "giao tiếp", ở đây nhắc đến việc giao tiếp qua những dòng code. Người ta có thể dễ dàng nói với một nhà phát triển khác trong nhóm, rằng "tôi sử dụng DP Command ở đây" và nhà phát triển khác không chỉ có thể hiểu về cách thiết kế mà còn có thể dễ dàng tìm ra lý do tại sao người kia lại thiết kế như vậy.
- Mẫu thiết kế thực sự giúp ích trong học tập. Đặc biệt. khi bạn là người mới trong một dự án. Bạn không tốn quá nhiều công sức để nghĩ về cách những người cũ đang làm với từng dòng code, thay vào đó bạn có thể hòa nhập nhanh hơn. Thêm vào đó, việc có một sự chỉ dẫn chuẩn mực (việc đi theo các DP) sẽ giúp bạn làm "đúng" được nhiều hơn.
- Ngoài ra, điều này giúp cung cấp cho các nhà phát triển có cái nhìn sâu sắc hơn về các phần của ứng dụng mà họ sử dụng của bên thứ 3 thiết kế.

Design Pattern giúp các nhà phát triển giảm đi sự khó khăn trong quá trình thiết kế hệ thống

- Phân tách hệ thống thành các đối tượng: Phần khó của Thiết kế hướng đối tượng là tìm ra các đối tượng phù hợp và phân tích để bóc tách một hệ thống. Trong mỗi bài toán, người thiết kế cần phải suy nghĩ về *tính đóng gói, độ chi*

tiết, tính phụ thuộc, linh hoạt, hiệu suất, khả năng mở rộng, tái sử dụng và nhiều hơn thế nữa. Việc phải làm hài hòa tất cả những tính chất trên gây nhiều khó khăn trong quá trình phân tích, bóc tách một vấn đề. DP thực sự giúp giảm bớt sự trừu tượng, khiến việc thiết kế trở nên "giảm phần nào" sự khó khăn.

- Chỉ ra rõ ràng việc triển khai một đối tượng: Chúng ta nên triển khai một đối tượng như thế nào? Với một Interface có thể có nhiều lớp cụ thể có thể triển khai nó, mỗi lớp có thể có các cách triển khai rất khác nhau. Các mẫu thiết kế cung cấp những hướng dẫn tổng quát để có thể dẫn đến một bản Code thực sự tốt.
- Đảm bảo cơ chế tái sử dụng: Khi nào nên sử dụng tính Kế thừa, khi nào sử dụng Kỹ thuật Tổng hợp (Composition), khi nào nên sử dụng các kiểu tham số? Làm thế nào để đưa ra được quyết định thiết kế đúng trong trường hợp này? Một lập trình viên, khi cố gắng thiết kế để đảm bảo có thể tái sử dụng, có khả năng duy trì của mã nguồn, họ gặp phải rất nhiều câu hỏi như trên. Việc có hiểu biết về các mẫu thiết kế có thể thực sự có ích khi đưa ra các quyết định như vậy.

Việc phát triển (mở rộng, bảo trì) hệ thống trở nên dễ dàng hơn

Mọi thứ đều có thể thay đổi, và việc phát triển phần mềm cũng không nằm ngoài quy luật trên, thậm chí việc thay đổi trong công nghệ còn diễn ra nhanh hơn nữa. Các thay đổi làm cho hệ thống phình to do các tính năng mới được thêm vào và bài toán hiệu năng cần được tối ưu.

Vậy làm thế nào để xây dựng phần mềm mà ảnh hưởng của những thay đổi này là nhỏ nhất? Việc hiểu được code (có thể được viết bởi người khác) đã khó và thay đổi code cũ mà không phát sinh các lỗi mới hoặc các bugs ko mong muốn lại càng khó khăn hơn.

Sẽ không có một kĩ thuật thần kỳ nào, nhưng việc dùng DP sẽ cung cấp các mẫu thiết kế có thể áp dụng vào thiết kế của bạn và giải quyết các vấn đề chung. Chúng không phải thư viện hay module. Chúng là những hướng dẫn để bạn tích hợp vào thiết kế để tạo nên các hệ thống hướng đối tượng linh hoạt và dễ bảo trì.

Để học được Design Pattern, bạn cần gì?

Cần khẳng định điều đầu tiên là DP không dành cho những bạn mới bắt đầu tìm hiểu về lập trình. Điều này không có nghĩa nếu bạn là người mới, bạn không được động vào DP, mà "dành cho" là khái niệm mà khi bạn tìm hiểu sâu, biết vận dụng nó trong các thiết kế của riêng mình.

Để tìm hiểu và học được Design Pattern thì bạn phải nắm chắc được kiến thức OOP đặc biệt là về abstract class, interface và static.

Các loại Design Pattern

Cơ bản, DP được chia làm 3 dạng chính, tổng cộng 32 mẫu designs:

Note: Các DP đánh dấu (*) là các DP quan trọng (theo các nguồn tham khảo)

Creational Pattern (nhóm khởi tạo):

Phục vụ trong việc khởi tạo đối tượng. Nhóm này gồm 9 mẫu design là:

- Abstract Factory.
- Prototype
- *Builder.
- Simple Factory
- *Factory Method.
- *Singleton
- Multiton. + Static Factory
- Pool.

Structural (nhóm cấu trúc):

Giúp thiết lập, định nghĩa quan hệ giữa các đối tượng. Nhóm này gồm 11 mẫu design là:

- *Adapter/ Wrapper.
- *Facade
- Bridge.
- Fluent Interface
- Composite.
- Flyweight
- Data Mapper.
- Registry
- Decorator.
- Proxy
- Dependency Injection.

Behavioral patterns (nhóm ứng xử):

Tập trung thực hiện các hành vi của đối tượng. Gồm 12 mẫu design là

- Chain Of Responsibilities.
- *Observer
- Command.
- Specification
- Iterator.
- *State
- Mediator.
- *Strategy
- Memento.
- Template Method
- Null Object.

- Visitor

Bạn có hứng thú với Design Pattern?

Với nhà thiết kế, để có được một thiết kế tốt nói chung và hiểu, vận dụng được Design Patterns nói riêng, họ sẽ cần thực tiễn công việc để có thể đi sâu vào các vấn đề và hiểu thông qua các ví dụ code cụ thể hơn là việc học lý thuyết.

Design Pattern đang dần trở nên vô cùng quan trọng. Nhưng việc học, hiểu và thành thạo được những mẫu thiết kế trên không phải là điều dễ dàng.

Nhưng trên con đường thành công, người có được sự đột phá sẽ trở nên khác biệt. Vậy tại sao không thử sức mình với Design Pattern, với việc nghiên cứu, nắm chắc lý thuyết, cùng với sự kiên trì thực hành, việc thành thạo được các mẫu thiết kế trên để vận dụng trong thiết kế của riêng bạn? Điều đó sẽ là điểm cộng vô cùng lớn cho bạn trong mắt những nhà tuyển dụng.

Vậy có nên thử?

Bài 3. Nhập môn Design Pattern theo phong cách kiếm hiệp

Nhập đề

Kinh thư ghi lại rằng, con đường tu chân có 3 cảnh giới: *Luyện khí, Trúc cơ và Kết đan*. Luyện khí là quá trình rèn thân luyện thể, cho phàm thân kiên cường dẻo dai. Trúc cơ là quá trình du nhập thiên địa linh khí vào thể nội, giúp khai thông kinh mạch. Khi thiên địa linh khí trong đan điền đạt tới một nồng độ nhất định, sẽ kết thành Kim Đan, đặt bước chân đầu tiên con đường tu chân đại đạo.

Con đường khởi đầu của code học cũng có 3 cảnh giới: Học đồ (Junior Developer), Học sĩ (Developer), Đại sư (Senior Developer). Để đạt đến cảnh giới Đại sư (senior), bất kì Học Sĩ (dev) nào cũng cần phải tường tận vài Design Pattern cơ bản để phòng thân. Bài viết này do tại hạ viết ra trong một phút cao hứng nhất thời, nhằm chia sẻ với các nhân sĩ võ lâm trên con đường truy cầu đại đạo.

Nhiều kẻ khi đạt đến cảnh giới Đại sư (Senior) cứ ngỡ rằng mình đã đạt đến cảnh giới tối cao của võ học mà không biết rằng "Thiên ngoại hữu thiên, nhân ngoại hữu nhân". Phía trên cảnh giới Đại sư còn có vô số cao thủ đạt tới những cảnh giới khác như Chưởng Môn (Project Manager) hoặc Tông sư (Software Architect). Những kẻ này hiếm thấy như phượng mao lân giác (lông phượng sừng lân), mang một thân võ công cao ngất ngưởng và lương cao ngất ngưỡng, lướt gió mà đi, đạp mây mà về. Do cảnh giới bản thân còn thấp, bần đạo tạm thời không bàn tới. Bằng hữu nào hứng thú có thể tìm hiểu thêm tại đây.

Hỏi thế gian DS là chi, mà bọn Dev thề nguyền sống chết

Nói một cách đơn giản, design pattern là **các mẫu thiết kế có sẵn, dung để giải quyết một vấn đề**. Áp dụng mẫu thiết kế này sẽ làm code dễ bảo trì, mở rộng hơn (Có thể sẽ khó hiểu hơn 1 chút). Nói văn hoa, design pattern là tinh hoa trong võ học, đã được các bậc tông sư đúc kết, truyền lưu từ đời này qua đời khác. Design pattern là **thiết kế dựa trên code**, nó nằm ở một cảnh giới cao hơn CODE, do đó đệ tử của bất kì môn phái nào (C#, Java, Python) cũng có thể áp dụng vào được. Ảnh lấy từ bí kíp Head First Design Pattern (xem phía dưới).

A Pattern is a solution to a problem in a context.

That's not the most revealing definition is it? But don't worry, we're going to step through each of these parts, context, problem and solution:

Example: You have a collection of objects.

The **context** is the situation in which the pattern applies. This should be a recurring situation.

The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.

You need to step through the objects without exposing the collection's implementation.

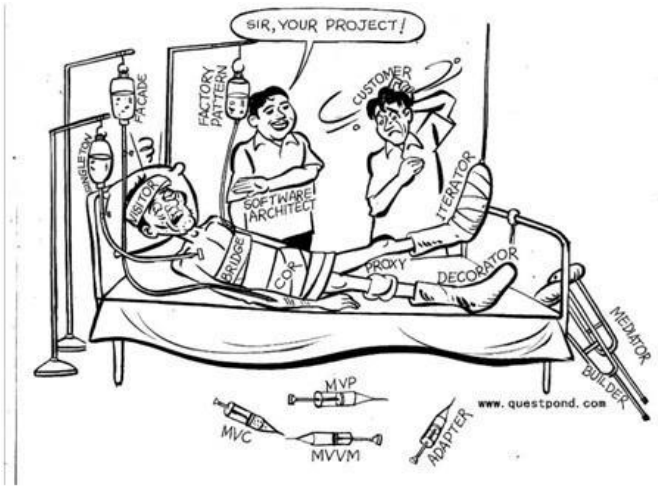
The **solution** is what you're after: a general design that anyone can apply which resolves the goal and set of constraints.

Encapsulate the iteration into a separate class.

This is one of those definitions that takes a while to sink in, but take it one step at a time. Here's a little mnemonic you can repeat to yourself to remember it:

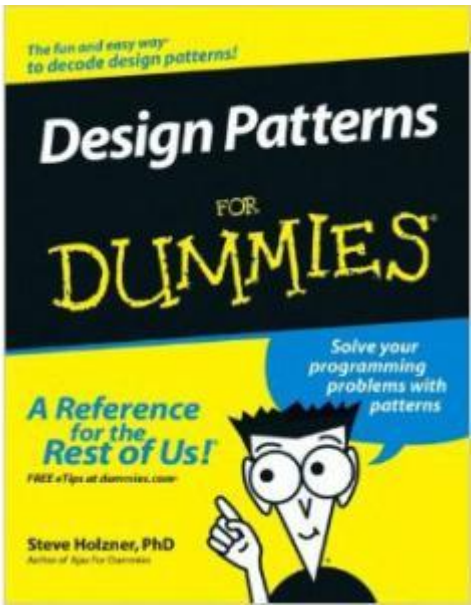
"If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution."

Trước khi dạy võ, các bậc đạo sư luôn dặn học trò rằng **học võ là để tu thân hành hiệp giúp đời**, không phải để ý vào một thân võ học mà đi bắt nạt kẻ yếu. Nay ta cũng có một lời khuyên tương tự: Học design pattern là để nâng cao trình độ, để **giải quyết vấn đề**, không phải đế lấy ra lòe thiên hạ. Nhiều kẻ học nghệ chưa tinh, ngựa non háu đá, nhét design pattern vào dự án một cách vô tội vạ, nhẹ thì tẩu hỏa nhập ma, võ công sụt giảm, nặng thì hồn phi phách tán, vĩnh kiếp không được siêu sinh. Các đạo hữu hãy nhìn kẻ than tàn ma dại phía dưới mà làm gương.



Design Pattern Kiếm Phổ

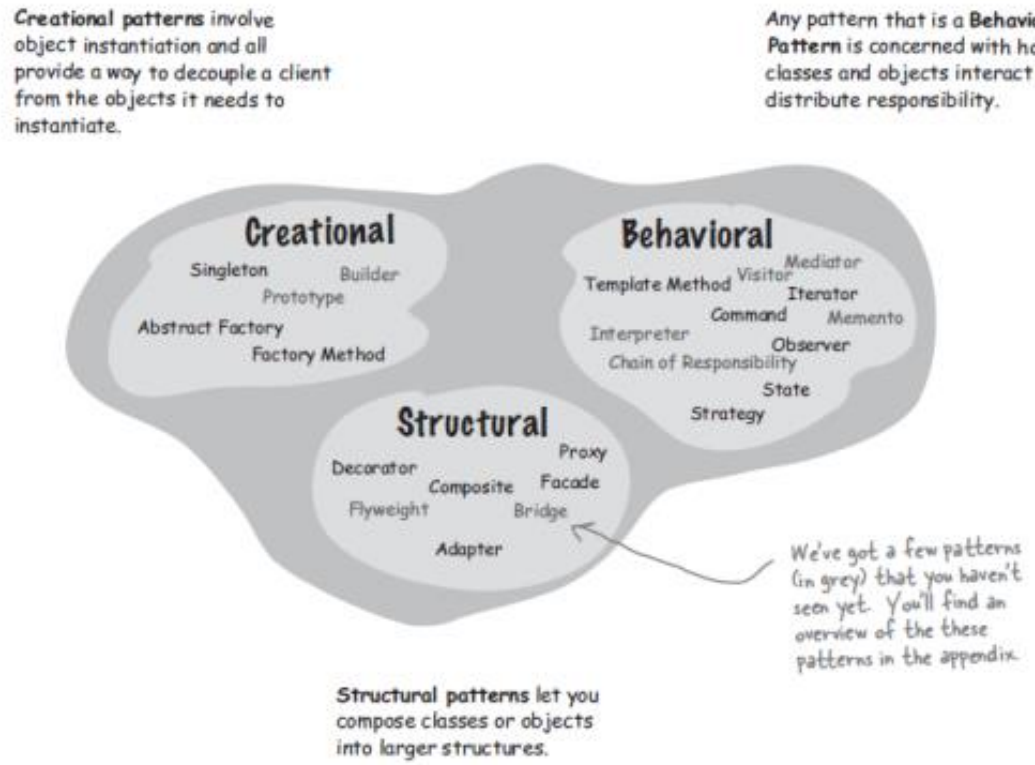
Bí kíp võ công đầu tiên nhắc đến design pattern là Design Patterns: Elements of Reusable Object-Oriented Software. Tuy nhiên, khẩu quyết trong quyển này khá khô cứng, khó truyền dạy, do đó các bậc cao nhân đã chỉnh sửa, xuất bản 2 cuốn bí kíp *dễ hiểu hơn* cho hậu thế là **Head First Design Patterns** và *Design Patterns For Dummies*. Thuở xưa khi đặt bước chân đầu tiên trên con đường cầu đạo-cạo đầu, bần đạo cũng tự tu luyện từ hai cuốn bí kiếp này. Các đạo hữu có thể lên mạng tải về ngâm cứu.



Khẩu quyết nhập môn Design Pattern

Có khá nhiều chiêu thức design pattern lưu lạc trong chốn giang hồ, song ta có thể tạm phân loại làm Tam Thức:

- **Khởi Thức (Creational Design Pattern):** Liên quan đến việc khởi tạo object. VD: Factory, Object Pool, Abstract Factory, Builder.
- **Cấu Thức (Structure Design Pattern):** Liên quan đến kết cấu, liên hệ giữa các object. VD: Adapter, Bridge, Decorator, Proxy, Composite, Facede.
- **Vi Thức (Behavioral Design Pattern):** Liên quan tới hành vi của các object. VD: Iterator, Mementor, Strategy, Template Method, Visitor.

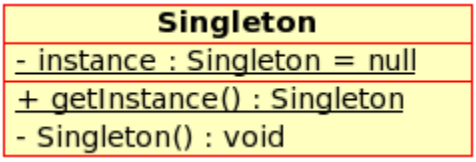


Khẩu quyết một chiêu thức Design Pattern thường có 3 phần. Khi muốn học một design pattern mới, hãy tập trung chú ý vào 3 phần này:

- **Thức Đề:** Vấn đề mà design pattern đó giải quyết
- **Thức Đồ:** Sơ đồ UML mô tả design pattern
- **Thức Phổ:** Code minh họa

Dưới đâu là một Design pattern đơn giản nhất mà hầu như học sĩ nào cũng biết: Đơn Thân Độc Mã, thuộc Khởi Thức, hay còn gọi là Singeton, thuộc loại Creational Design Pattern.

- **Thức Đề:** Design Pattern này được dùng khi ta muốn đảm bảo chỉ có duy nhất một object được sinh ra trong toàn hệ thống.
- **Thức Đồ:**



- **Thức Phổ:**

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
```

```
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Thay lời kết

Xin nhắc lại một lần nữa: *Design Pattern được tạo ra để giải quyết vấn đề, chứ không phải để phức tạp hóa nó*. Các bậc cao nhân có câu: nước có thể dâng thuyền, cũng có thể lật thuyền. Design Pattern có thể giải quyết vấn đề, cũng có thể làm nó rắc rối phức tạp hơn.

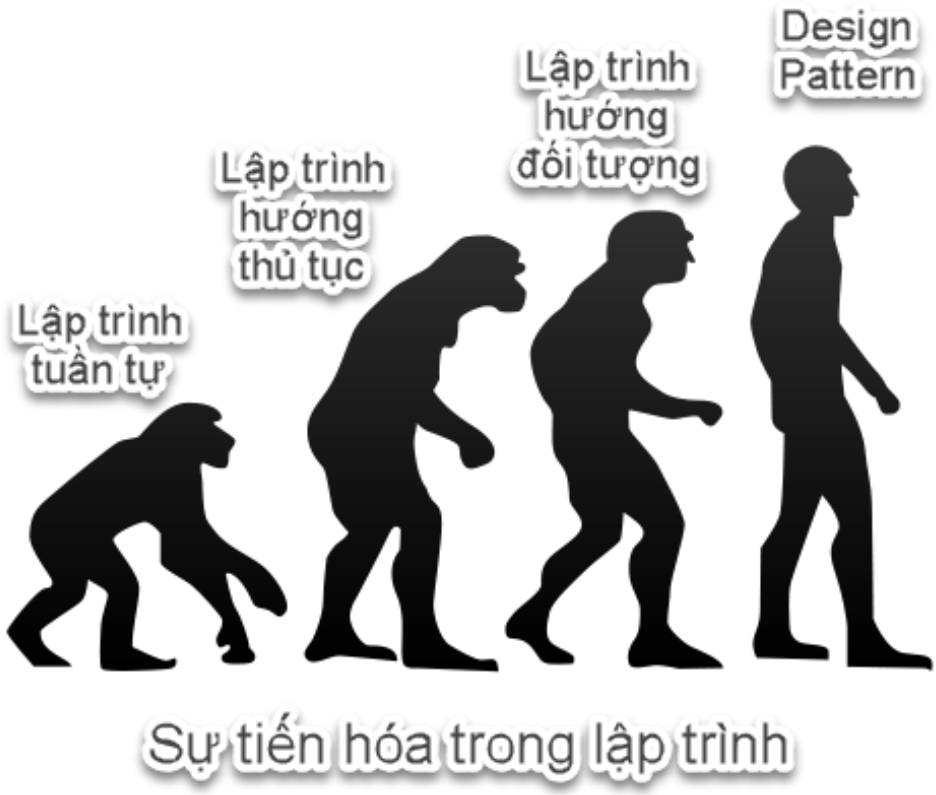
Kẻ sĩ dùng design pattern cũng chia làm ba cảnh giới. Kẻ sơ nhập thì nhìn đâu cũng thấy pattern, chỉ lo áp dụng, nhét rất nhiều pattern vào mà không quan tâm đến thiết kế. Lăn lộn giang hồ một thời gian, đến cảnh giới cao thủ, sẽ học được rằng khi nào cần dùng pattern, khi nào không. Đến cấp bậc đại sư, chỉ dùng pattern khi đã rõ lợi hại của nó, biết lấy sự đơn giản hài hòa của design tổng thể làm trọng. Có thể tổng kết quá trình này bằng một câu:

"Khi chưa học đạo, ta thấy núi là núi, sông là sông. Khi mới học đạo, ta thấy núi không phải là núi, sông không phải là sông. Sau khi học đạo, ta lại thấy núi chỉ là núi, sông chỉ là sông."

Bài 4: Design Pattern sự tiến hóa trong lập trình

PHP là ngôn ngữ lập trình hướng đối tượng rất tốt, các bài toán được giải quyết bằng tư duy hướng đối tượng đã giải quyết được nhiều vấn đề trong duy trì và phát triển ứng dụng. Tuy nhiên, chỉ hướng đối tượng thôi chúng ta mới xử lý được ở mức vi mô, giống như việc xây một ngôi nhà, các đối tượng như gạch, vữa, đá ốp lát... thì việc xây dựng vẫn rất lâu, chúng ta cần tạo sẵn ra những bức tường, cột bê tông... để chỉ cần lắp ghép rất nhanh để ra được ngôi nhà. Design Pattern chính là công việc tạo ra các mẫu tường, mẫu cột bê tông... nó giải quyết được các vấn đề có sẵn và xây dựng sẽ nhanh chóng hơn.

Trong quá trình xây dựng ra các khuôn mẫu, chúng ta cũng có thể áp dụng các nguyên lý thiết kế hướng đối tượng SOLID, nắm vững cả các nguyên lý và cách thức áp dụng mẫu thiết kế giúp cho lập trình hướng đối tượng được nâng lên một tầm cao mới.



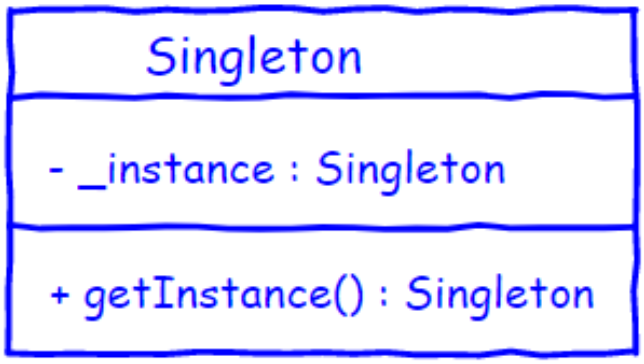
Design Pattern có thể thấy là một cấp độ khác trong lập trình ứng dụng, nó là các mẫu thiết kế có sẵn được đúc kết bởi rất nhiều các lập trình viên kinh nghiệm nhằm giải quyết các vấn đề chung. Áp dụng các mẫu có sẵn này vào lập trình giúp ứng dụng dễ duy trì và cập nhật, tuy nhiên nó cũng làm cho những ai không am hiểu về Design Pattern cảm thấy lúng túng. Design Pattern là kiến thức có thể dùng chung cho nhiều các ngôn ngữ hướng đối tượng khác như C#, Java..., chúng được phân ra làm nhiều loại theo chức năng:

- **Creational Design Pattern**: các mẫu này chuyên sử dụng cho khởi tạo đối tượng có thể kể đến như Singleton, Factory, Builder, Prototype...
- **Strutural Design Pattern**: các mẫu liên quan đến cấu trúc, kết cấu các đối tượng, ví dụ Composite, Decorator, Facade, Adapter, Proxy...
- **Behavioral Design Pattern**: các mẫu giải quyết các vấn đề về hành vi đối tượng như Strategy, Iterator, State, Observer...
- **Architectural Design Pattern**: các mẫu liên quan đến kiến trúc ứng dụng: MVC, SOA (Service-Oriented Architecture), microservice...

Tổng cộng có khoảng hơn 20 Design Pattern được áp dụng trong lập trình hướng đối tượng, để có thể nắm bắt được hết cũng mất kha khá thời gian. Trong khuôn khổ bài viết, bạn sẽ làm quen với một số pattern tiêu biểu, qua đó nắm vững được **Design Pattern là gì?** và **Cách áp dụng Design Pattern trong phát triển ứng dụng**.

1. Singleton Pattern

Tần suất sử dụng: 4/5. Singleton Pattern được sử dụng khá nhiều trong lập trình.



Singleton pattern thuộc về Creational Design Pattern là một mẫu áp dụng cho việc khởi tạo đối tượng, áp dung pattern này khi ứng dụng của bạn muốn tạo ra một thực thể duy nhất từ một class và dùng chung nó cho nhiều trường hợp. Ví dụ, website cần một đối tượng kết nối đến database nhưng chỉ cần duy nhất một đối tượng cho toàn bộ ứng dụng, sử dụng Singleton Pattern sẽ giải quyết được vấn đề này. Để bắt đầu bạn sử dụng một thuộc tính static để đảm bảo rằng chỉ có một thực thể của lớp này tồn tại.

```
class SomeClass {
  static private $_instance = NULL;
}
```

Trong phần lập trình hướng đối tượng, thuộc tính static được chia sẻ giữa các đối tượng của class, do đó nếu đã có một thực thể của class thì tất cả sẽ tham chiếu đến class đó có thể sử dụng thuộc tính này. Bước tiếp theo là tạo ra một phương thức sẽ tạo ra một instance của class nếu nó chưa tồn tại và trả về instance đó.

```
class SomeClass {
  static private $_instance = NULL;
  static function getInstance() {
    if (self::$_instance == NULL) {
      self::$_instance = new SomeClass();
    }
    return self::$_instance;
  }
}
```

Singleton pattern thường sử dụng một phương thức với cái tên getInstance() để kiểm tra thuộc tính $_instance, nếu nó có giá trị NULL thì sẽ tạo ra một instance và gán cho thuộc tính này, kết quả là một instance được trả về. Như vậy, class này có thể được sử dụng như sau:

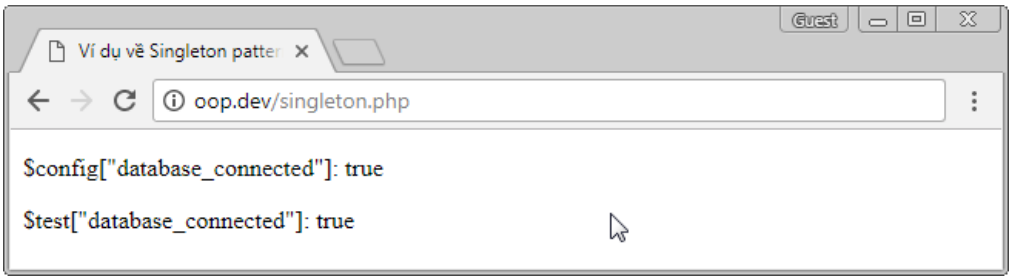
```
$obj1 = SomeClass::getInstance();
```

Trong đối tượng đầu tiên của ứng dụng, thực thể được tạo ra và được gán cho thuộc tính private bên trong nó. Chúng ta cùng bắt đầu với ví dụ về Singleton pattern, tạo ra một file để thiết lập cấu hình. Bạn có thể tham khảo cách tạo môi trường thực hành với PHP giúp thực hiện các đoạn code test dễ dàng hơn. Tạo một file là singleton.php trong thư mục oop/pattern với nội dung

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
  <title>Ví dụ về Singleton pattern</title>
</head>
<body>
    <?php
    /* The Config class.
     * The class contains two attributes: $_instance and $settings.
     * The class contains four methods:
     * - __construct()
     * - getInstance()
     * - set()
     * - get()
     */
    class Config {

       static private $_instance = NULL;
       private $_settings = array();

       // Private methods cannot be called
       private function __construct() {}
       private function __clone() {}

    // Phương thức này trả về một thực thể của class
       static function getInstance() {
  if (self::$_instance == NULL) {
             self::$_instance = new Config();
          }
          return self::$_instance;
       }
    // Phương thức này thiết lập cấu hình
       function set($index, $value) {
          $this->_settings[$index] = $value;
       }
    // Phương thức này lấy thiết lập cấu hình
       function get($index) {
          return $this->_settings[$index];
       }
    }
    // Tạo một đối tượng Config
    $config = Config::getInstance();
```

```
    // Thiết lập các giá trị trong thuộc tính cấu hình
    $config->set('database_connected', 'true');

    // In giá trị cấu hình
    echo '<p>$config["database_connected"]: ' . $config
>get('database_connected') . '</p>';

    // Tạo một đối tượng thứ hai
    $test = Config::getInstance();
    echo '<p>$test["database_connected"]: ' . $test-
>get('database_connected') . '</p>';

    // Xóa các đối tượng sau khi sử dụng
    unset($config, $test);
    ?>
</body>
</html>
```

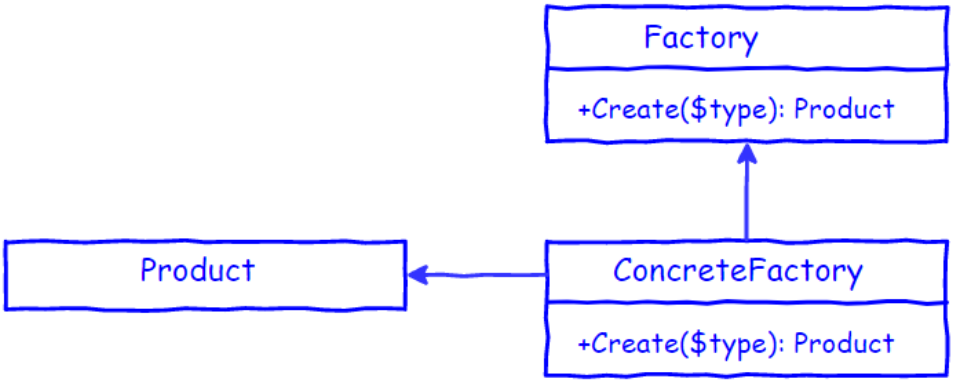
Truy cập thử vào file này chúng ta thấy kết quả như sau:



Bạn thấy đấy, mỗi khi tạo ra một đối tượng của class Config nó sẽ kiểm tra và chỉ tạo thực thể (instance) khi chưa có thực thể nào được tạo. Biến $test sẽ cùng tham chiếu đến đối tượng từ class Config do đó các thiết lập cấu hình có thể chia sẻ lại.

2. Factory Pattern

Tần suất sử dụng: 5/5, Factory Pattern được sử dụng cực nhiều trong lập trình.



Factory pattern cũng như Singleton pattern thuộc về dạng Creational Design Pattern, tuy nhiên có một chút khác biệt. Singleton pattern áp dụng để tạo và quản lý một đối tượng duy nhất của một class trong khi Factory pattern được sử dụng để có thể tạo ra nhiều đối tượng khác nhau từ nhiều class. Tại sao cần Factory pattern trong khi chúng ta có thể tạo được đối tượng từ Singleton Pattern? Vấn đề là ở chỗ đôi khi chúng ta không biết trước được là muốn tạo đối tượng từ class cụ thể nào, do đó việc chỉ định class để tạo ra đối tượng cần phải được gán động. Factory pattern cần sử dụng một lớp trìu tượng (abstract class) và có một phương thức static, được quy ước tên là Create(), factory(), factoryMethod() hoặc createInstance(). Phương thức này có một tham số để nhận biết dạng đối tượng cần tạo và trả về đối tượng này.

```
static function Create($type) {
  // Kiểm tra tham số $type và tạo đối tượng từ class tương ứng để
trả về.
  return new SomeClassType();
}
```

Có vẻ chưa hiểu lắm, không sao, ví dụ tiếp theo sẽ giúp bạn hiểu ngay thôi. Chúng ta quay trở lại ví dụ về các hình chữ nhật (rectangle), hình tam giác (triangle) (Xem Phần 3 của Lập trình hướng đối tượng trong PHP, ví dụ đầu tiên). Chúng ta tạo ra file factory.php trong thư mục oop\pattern với nội dung như sau:

```
<!doctype html>
<html lang="vi">
<head>
    <meta charset="utf-8">
  <title>Ví dụ về Factory pattern</title>
</head>
<body>
    <?php
    #------------ ĐỊNH NGHĨA CLASS ----------------------#
    /* Định nghĩa class ShapeFactory sử dụng Factory pattern
     * The class contains no attributes.
     * The class contains one method: Create().
     */
abstract class ShapeFactory {
        // Phương thức static để tạo đối tượng
        static function Create($type, array $sizes) {
            // Xác định dạng đối tượng theo tham số nhận vào
            switch ($type) {
                case 'rectangle':
                    return new Rectangle($sizes[0], $sizes[1]);
                    break;
                case 'triangle':
                    return new Triangle($sizes[0], $sizes[1],
$sizes[2]);
                    break;
            }
```

```
        }
    }

    /* Định nghĩa lớp trìu tượng Shape
   * Lớp Shape không có thuộc tính
   * Lớp Shape có 2 phương thức trìu tượng:
     * - getArea()
     * - getPerimeter()
    */
    abstract class Shape {
        abstract protected function getArea();
        abstract protected function getPerimeter();
    }

    /* Định nghĩa lớp Triangle
 * Lớp Triangle có 2 thuộc tính:
     * - private $_sides (array)
     * - private $_perimeter (number)
   * Lớp Triangle có 3 phương thức:
     * - _ _construct()
     * - getArea()
     * - getPerimeter()
     */
    class Triangle extends Shape {
        private $_sides = array();
        private $_perimeter = NULL;
        function __construct($s0 = 0, $s1 = 0, $s2 = 0) {
            $this->_sides[] = $s0;
            $this->_sides[] = $s1;
            $this->_sides[] = $s2;

      // Tính toán và thiết lập chu vi hình tam giác
            $this->_perimeter = array_sum($this->_sides);
        }
    // Phương thức tính diện tích hình tam giác từ chu vi và các cạnh
        public function getArea() {
            return (SQRT(($this->_perimeter/2) * (($this-
>_perimeter/2) - $this->_sides[0]) * (($this->_perimeter/2) - $this-
>_sides[1]) * (($this->_perimeter/2) - $this->_sides[2])));
        }
    // Phương thức lấy chu vi hình tam giác
        public function getPerimeter() {
            return $this->_perimeter
        }
    }

    /* Định nghĩa class Rectangle
   * Các thuộc tính của class: width(chiều rộng), height(chiều cao).
   * Các phương thức của lớp:
     * - setSize()
     * - getArea()
     * - getPerimeter()
```

```
 * - isSquare()
 */
 class Rectangle {
// Khai báo các thuộc tính
     public $width = 0;
     public $height = 0;

// Hàm khởi tạo
     function __construct($w = 0, $h = 0) {
        $this->width = $w;
        $this->height = $h;
     }

// Phương thức này thiết lập các kích thước của hình chữ nhật
     function setSize($w = 0, $h = 0) {
         $this->width = $w;
         $this->height = $h;
     }
// Phương thức này tính diện tích hình chữ nhật
     function getArea() {
         return ($this->width * $this->height);
     }

// Phương thức này tính chu vi hình chữ nhật
     function getPerimeter() {
         return ( ($this->width + $this->height) * 2 );
     }

// Phương thức này kiểm tra xem hình chữ nhật này có phải là hình vuông
     function isSquare() {
         if ($this->width == $this->height) {
    return true; // Hình chữ nhật
         } else {
    return false; // Không phải hình chữ nhật
         }
     }
 }
 #------------ KẾT THÚC ĐỊNH NGHĨA CLASS ----------------------#

 if (isset($_GET['shape'], $_GET['dimensions'])) {
     // Tạo ra một đối tượng từ với thông số từ query string
     $obj = ShapeFactory::Create($_GET['s'], $_GET['d']);

 echo "<h2>Tạo ra hình {$_GET['shape']}:</h2>";
 echo '<p>Diện tích hình: ' . $obj->getArea() . '</p>';
echo '<p>Chu vi hình: ' . $obj->getPerimeter() . '</p>';
 } else {
 echo '<p>Cần cung cấp hình dạng và kích thước!</p>';
 }

 // Xóa đối tượng
 unset($obj);
```

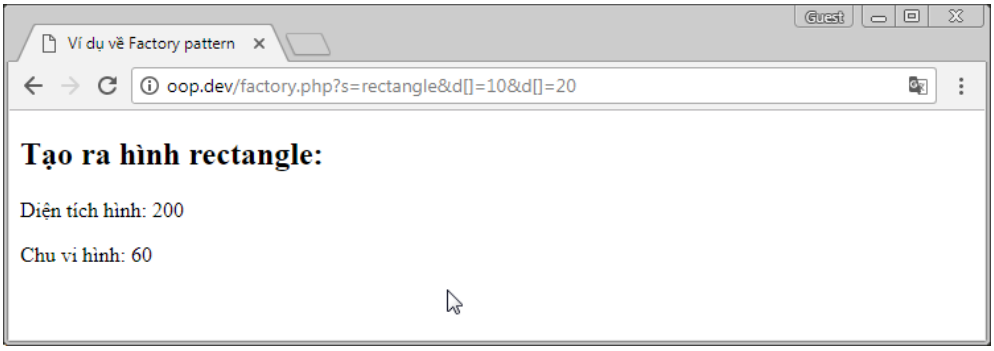
```
    ?>
</body>
</html>
```

Trong ví dụ này chúng ta đã tạo ra lớp ShapeFactory áp dụng Factory Pattern, nó sẽ nhận các tham số đầu vào để tạo ra một thực thể tương ứng với class cần tạo. Tiếp đó là các class Rectangle (hình chữ nhật) và Triangle (hình tam giác) được mở rộng từ lớp trìu tượng Shape (hình). Tiếp đó, chúng ta sử dụng các query string của URL để đưa vào dạng hình và kích thước hình cần tạo:

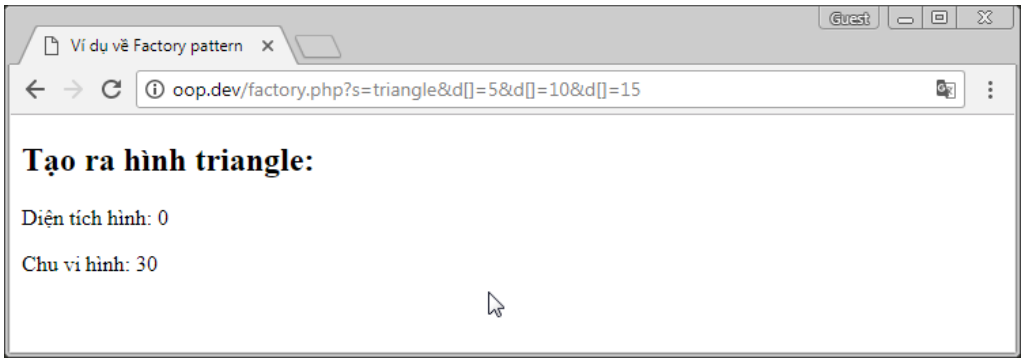
- s - shape là hình cần tạo: rectangle hoặc triangle
- d - dimension là kích thước của hình, nếu là tam giác nhập 3 cạnh, nếu là hình chữ nhật nhập 2 cạnh.

Thực hiện chạy thử

http://oop.dev/factory.php?s=rectangle&d[]=10&d[]=20 chúng ta được kết quả:



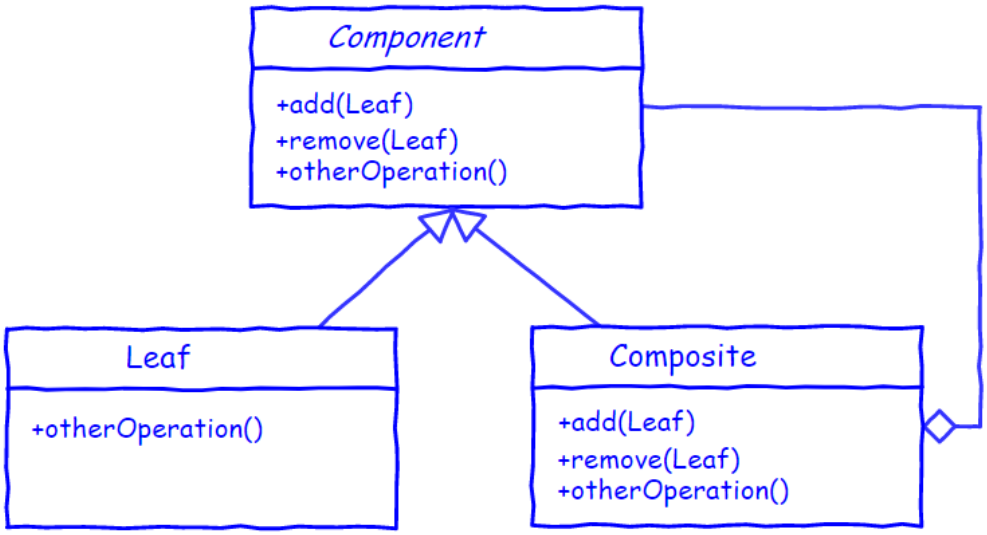
Tiếp theo thử tạo hình tam giác với các cạnh là 5, 10, 15 với URL http://oop.dev/factory.php?s=triangle&d[]=5&d[]=10&d[]=15 kết quả như sau:



Như vậy với việc áp dụng Factory Pattern chúng ta đã tạo được các đối tượng khác nhau tùy thuộc vào từng tình huống mà không biết trước được như Singleton Pattern.

3. Composite pattern

Tần suất sử dụng 4/5, Composite pattern được sử dụng khá nhiều.



Hai pattern đầu thuộc về Creational Design Pattern, tiếp theo chúng ta sẽ làm quen với Composite Pattern nó là Structural Design Pattern, là mẫu thiết kế liên quan đến cấu trúc, kết cấu của các đối tượng. Nó được áp dụng để cấu trúc một class theo tiêu chuẩn hoặc điều chỉnh cấu trúc một class đang tồn tại. Trong thực tế, một form HTML có thể chứa một hoặc nhiều các thành phần form, mỗi thành phần này sẽ có cùng các hành vi như hiển thị, kiểm tra dữ liệu, hiển thị lỗi… Nếu không áp dụng các mẫu thì chúng ta sẽ lặp đi lặp lại code rất nhiều và giải pháp cho vấn đề này là ứng dụng Composite pattern. Chúng ta tạo ra một abstract class:

```
abstract class FormComponent {
  abstract function add (FormComponent $obj);
abstract function remove (FormComponent $obj);
  abstract function display();
  abstract function validate();
  abstract function showError();
}
```

Lớp trìu tượng ở trên có sử dụng type hinting (cách xác định dạng dữ liệu cho tham số), hai phương thức đầu chính là cách mà Composite pattern sử dụng. Mỗi lớp con sẽ kế thừa từ lớp cha và nó cần phải định nghĩa các phương thức trìu tượng được implement từ lớp cha trìu tượng. Ví dụ:

```
class Form extends FormComponent {
  private $_elements = array();
  function add(FormComponent $obj) {
    $this->_elements[] = $obj;
  }
  function display() {
    // Display the entire form.
  }
}
class FormElement extends FormComponent {
  function add(FormComponent $obj) {
    return $obj; // Or false.
```

```
  }
  function display() {
    // Display the element.
  }
}
```

Class Form định nghĩa

phương thức add() được implement từ FormConponent, nó cho phép bạn thêm thành phần vào form:

```
$form = new Form();
$email = new FormElement();
$form->add($email);
```

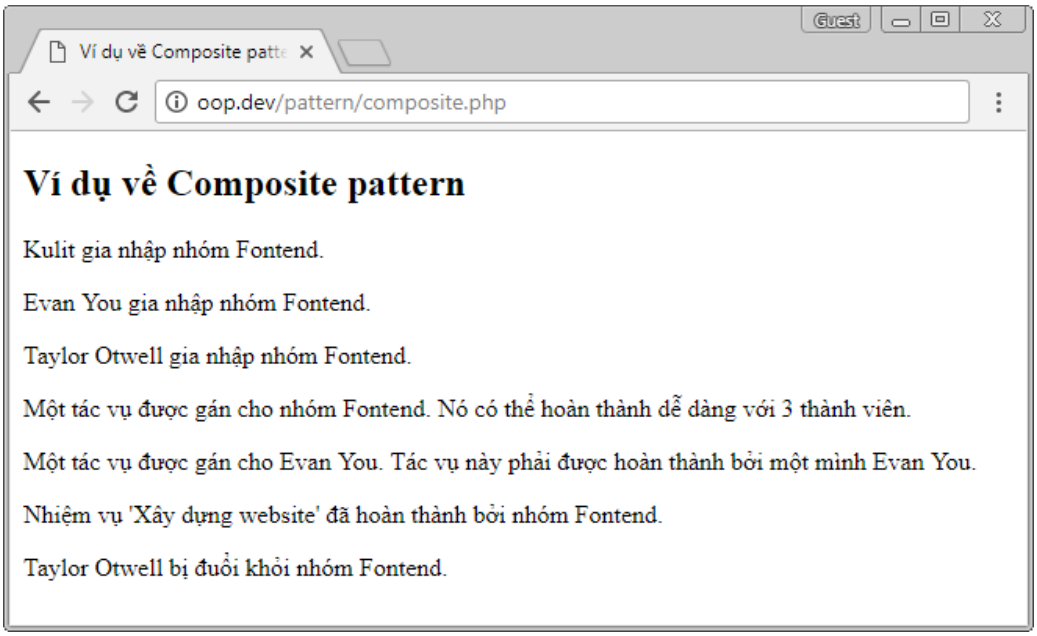
Chú ý là FormElement cũng định nghĩa phương thức add(), nhưng phương thức này không làm gì cả, vì chúng ta không cần thêm thành phần form vào một thành phần form. Thay vào đó, phương thức add() này trả về đối tượng được thêm vào hoặc trả về một giá trị hoặc bung ra một lỗi. Với ví dụ ở dạng mẫu trên, chúng ta vẫn chưa thật sự hiểu rõ về Composite pattern. Một ví dụ cụ thể tiếp theo sẽ giúp bạn hiểu chi tiết. Ví dụ dưới đây về một ứng dụng quản lý các công việc cần làm của một nhóm và của từng thành viên trong nhóm.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
  <title>Ví dụ về Composite pattern</title>
</head>
<body>
  <h2>Ví dụ về Composite pattern</h2>
    <?php
    #------------ ĐỊNH NGHĨA CLASS ----------------------#
    /* Định nghĩa class WorkUnit sử dụng Composite pattern
   * Lớp có 2 thuộc tính: tasks, name.
   * Lớp có 5 phương thức: __construct(), getName(), add(), remove(), assignTask(),
completeTask().
*/
    abstract class WorkUnit {
    // Các tác vụ cần làm
        protected $tasks = array();
    // Lưu tên nhân viên hoặc tên nhóm
        protected $name = NULL;

        function __construct($name) {
            $this->name = $name;
        }
        function getName() {
            return $this->name;
        }
    // Các phương thức trìu tượng cần thực hiện
        abstract function add(Employee $e);
```

```
        abstract function remove(Employee $e);
        abstract function assignTask($task);
        abstract function completeTask($task);
    }

  /* Lớp Team mở rộng từ lớp WorkUnit.
   * Lớp có 1 thuộc tính: _employees.
   * Lớp có 1 phương thức: getCount().
      */
    class Team extends WorkUnit {
    // Lưu các thành viên của nhóm
        private $_employees = array();
    // Thực hiện các phương thức trìu tượng
        function add(Employee $e) {
            $this->_employees[] = $e;
      echo "<p>{$e->getName()} gia nhập nhóm {$this->getName()}.</p>";
        }
        function remove(Employee $e) {
            $index = array_search($e, $this->_employees);
            unset($this->_employees[$index]);
            echo "<p>{$e->getName()} bị đuổi khỏi nhóm {$this-
>getName()}.</p>";
        }
        function assignTask($task) {
            $this->tasks[] = $task;
            echo "<p>Một tác vụ được gán cho nhóm {$this-
>getName()}. Nó có thể hoàn thành dễ dàng với {$this->getCount()}
thành viên.</p>";
        }
        function completeTask($task) {
            $index = array_search($task, $this->tasks);
            unset($this->tasks[$index]);
            echo "<p>Nhiệm vụ '$task' đã hoàn thành bởi nhóm {$this-
>getName()}.</p>";
        }
    // Phương thức trả về số thành viên trong nhóm
        function getCount() {
            return count($this->_employees);
        }
    }
 /* Lớp Employee mở rộng từ lớp WorkUnit


   * Lớp không có thuộc tính và phương thức nào
      */
    class Employee extends WorkUnit {
        // Empty functions
        function add(Employee $e) {
            return false;
        }
        function remove(Employee $e) {
            return false;
```

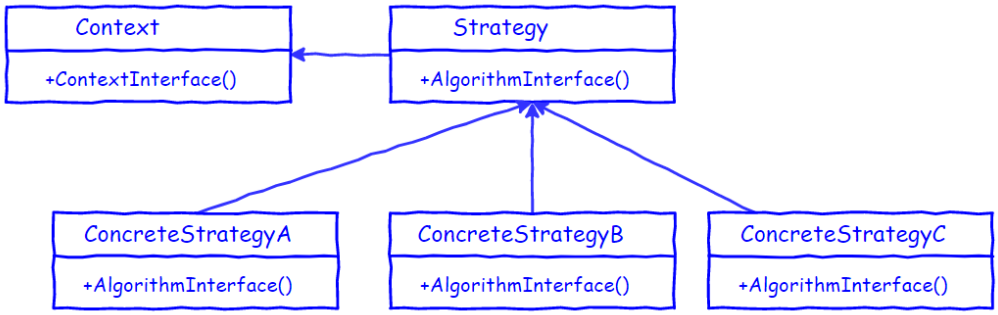
```
        }

    // Thực hiện phương thức trìu tượng
        function assignTask($task) {
            $this->tasks[] = $task;
            echo "<p>Một tác vụ được gán cho {$this->getName()}. Tác
vụ này phải được hoàn thành bởi một mình {$this->getName()}.</p>";
        }
        function completeTask($task) {
            $index = array_search($task, $this->tasks);
            unset($this->tasks[$index]);
            echo "<p>Nhiệm vụ '$task' được hoàn thành bởi {$this-
>getName()}. </p>";
        }

    }
    #------------ KẾT THÚC ĐỊNH NGHĨA CLASS ----------------------#

    // Tạo đối tượng
    $fontend = new Team('Fontend');
 $kulit = new Employee('Kulit');
$evan = new Employee('Evan You');
    $taylor = new Employee('Taylor Otwell');

    // Gán nhân viên vào nhóm fontend
    $fontend->add($kulit);
    $fontend->add($evan);
    $fontend->add($taylor);

  // Gán các tác vụ cho nhóm và nhân viên
  $fontend->assignTask('Xây dựng website');
  $evan->assignTask('Xây dựng fontend');
  // Hoàn thành một tác vụ
  $fontend->completeTask('Xây dựng website');

  // Chuyển Taylor Otwell sang nhóm backend
    $fontend->remove($taylor);

    // Xóa các đối tượng
    unset($fontend, $kulit, $evan, $taylor);
    ?>
</body>
</html>
```

Trong ví dụ này có nhóm và nhân viên, nhóm cũng có công việc của nhóm và nhân viên có công việc của nhân viên. Áp dụng Composite Pattern giúp chúng ta nhìn nhận Nhóm cũng giống như Nhân viên, và các xử lý trên Nhóm và Nhân viên là tương tự nhau. Kết quả khi chạy file composite.php như sau:



4. Strategy pattern

Tần suất sử dụng: 4/5, Strategy pattern được sử dụng khá nhiều trong lập trình.



Mỗi dạng Design Pattern sẽ giới thiệu một Pattern tiêu biểu và Behavioral Design Pattern sẽ được bàn luận với pattern cuối cùng là Strategy pattern, mẫu này áp dụng khi làm việc với các hành vi của đối tượng, nó "đánh dấu" cách ứng dụng chạy. Factory pattern có thể thay đổi dạng đối tượng thì Strategy có thể thay đổi thuật toán (hành vi) của đối tượng. Strategy rất hữu ích trong một số trường hợp nơi mà các class là tương tự nhau nhưng không liên quan và khác nhau về hành vi. Ví dụ, chúng ta cần lọc các chuỗi, các bộ lọc khác nhau có thể sử dụng:

- Loại bỏ các thẻ HTML
- Loại bỏ các từ ngữ không phù hợp.
- Loại bỏ các ký tự sử dụng để gửi thư rác thông qua hình thức liên lạc

Thông thường chúng ta sẽ làm 3 giải pháp và áp dụng chúng vào các chuỗi cần lọc. Các bộ lọc có thể được áp dụng một cách khác nhau. Đầu tiên, định nghĩa interface với các tính năng cần thiết

```
interface Filter {
```

```
  function filter($str);
}
```

Xác định dạng bộ lọc sau đó implement các phiên bản thích hợp của phương thức trong interface:

```
class HtmlFilter implements Filter {
  function filter($str) {
  // Loại bỏ mã HTML
    return $str;
  }
}
class SwearFilter implements Filter {
  function filter($str) {
  // Loại bỏ các từ ngữ không phù hợp.
    return $str;
  }
}
```

Cuối cùng, sử dụng bộ lọc trong một class khác.

```
class FormData {
  private $_data = NULL;
  function __construct($input) {
    $this->_data = $input;
  }
  function process(Filter $type) {
    $this->_data = $type->filter($this->_data);
  }
}
```

Phương thức process() nhận một đối tượng dạng Filter và thông qua đó dữ liệu được lọc.

```
$form = new FormData($someUserInput);
if (!$allowHTML) {
  $form->process(new HtmlFilter());
}
if (!allowSwear) {
  $form->process(new SwearFilter());
}
```

OK, lý thuyết là như vậy, chúng ta cùng áp dụng vào thực tế, ví dụ về một ứng dụng quản lý sinh viên. Tạo ra file strategy.php trong oop\pattern với nội dung:

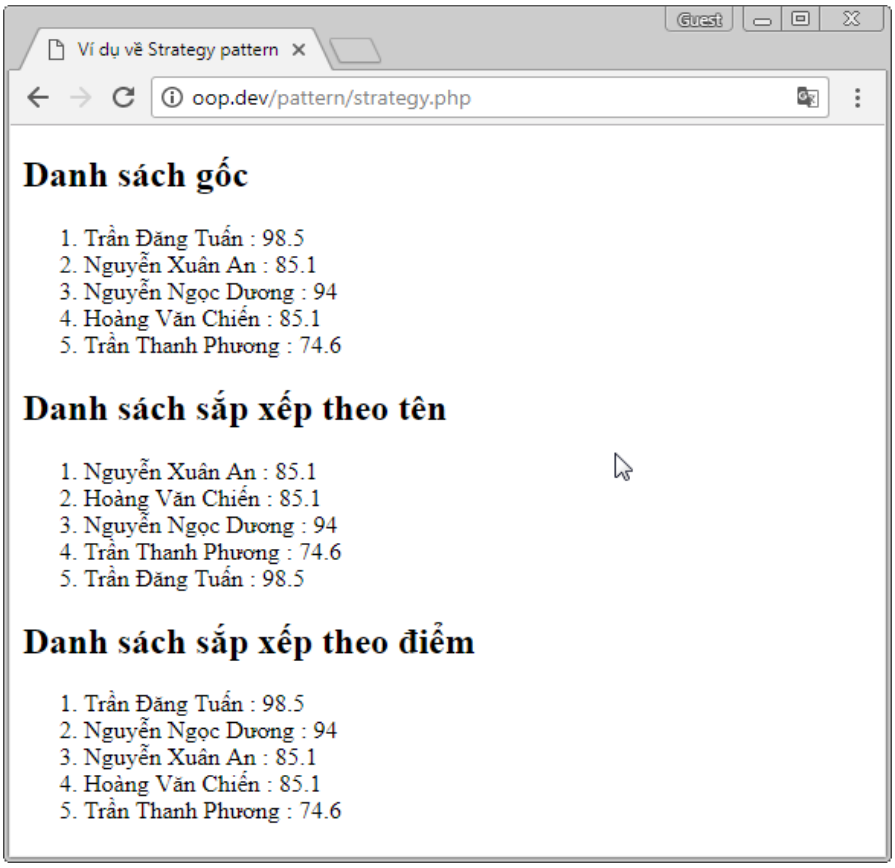
```
<!doctype html>
<html lang="vi">
<head>
```

```
    <meta charset="utf-8">
  <title>Ví dụ về Strategy pattern</title>
</head>
<body>
    <?php
    // Interface Sort định nghĩa phương thức sort()
    interface iSort {
        function sort(array $list);
    }
    // Lớp MultiAlphaSort sắp xếp mảng đa chiều chứa ký tự
    class MultiAlphaSort implements iSort {
        // Cách sắp xếp: tăng dần, giảm dần
        private $_order;

        // Sort index:
        private $_index;

        function __construct($index, $order = 'ascending') {
            $this->_index = $index;
            $this->_order = $order;
        }

    // Phương thức thực hiện sắp xếp
        function sort(array $list) {

            // Change the algorithm to match the sort preference:
            if ($this->_order == 'ascending') {
                uasort($list, array($this, 'ascSort'));
            } else {
                uasort($list, array($this, 'descSort'));
            }
            return $list;
        }
    // Phương thức so sánh hai giá trị
        function ascSort($x, $y) {
            return strcasecmp($x[$this->_index], $y[$this->_index]);
        }
        function descSort($x, $y) {
            return strcasecmp($y[$this->_index], $x[$this->_index]);
        }
    }

    // Class MultiNumberSort sắp xếp một mảng đa chiều
    class MultiNumberSort implements iSort {
    // Cách sắp xếp
        private $_order;

        // Sort index
        private $_index;

        function __construct($index, $order = 'ascending') {
            $this->_index = $index;
```

```
            $this->_order = $order;
        }

    // Thực hiện sắp xếp
        function sort(array $list) {
            // Thay đổi thuật toán phù hợp với thiết lập
            if ($this->_order == 'ascending') {
                uasort($list, array($this, 'ascSort'));
            } else {
                uasort($list, array($this, 'descSort'));
            }
            return $list;
        }
    // Phương thức so sánh hai giá trị
        function ascSort($x, $y) {
            return ($x[$this->_index] > $y[$this->_index]);
        }
        function descSort($x, $y) {
            return ($x[$this->_index] < $y[$this->_index]);
        }
    }

/* Lớp StudentsList
 * Lớp có 1 thuộc tính: _students.
 * Lớp có 3 phương thức:
   * - __construct()
   * - sort()
   * - display()
   */
   class StudentsList {
        // Danh sách sinh viên được sắp xếp
        private $_students = array();

        function __construct($list) {
            $this->_students = $list;
        }

    // Thực hiện sắp xếp sử dụng một thực thi từ iSort
        function sort(iSort $type) {
            $this->_students = $type->sort($this->_students);
        }

    // Hiển thị danh sách sinh viên dạng HTML
        function display() {
            echo '<ol>';
            foreach ($this->_students as $student) {
                echo "<li>{$student['last_name']}
{$student['first_name']} : {$student['grade']}</li>";
            }
            echo '</ol>';
        }
```

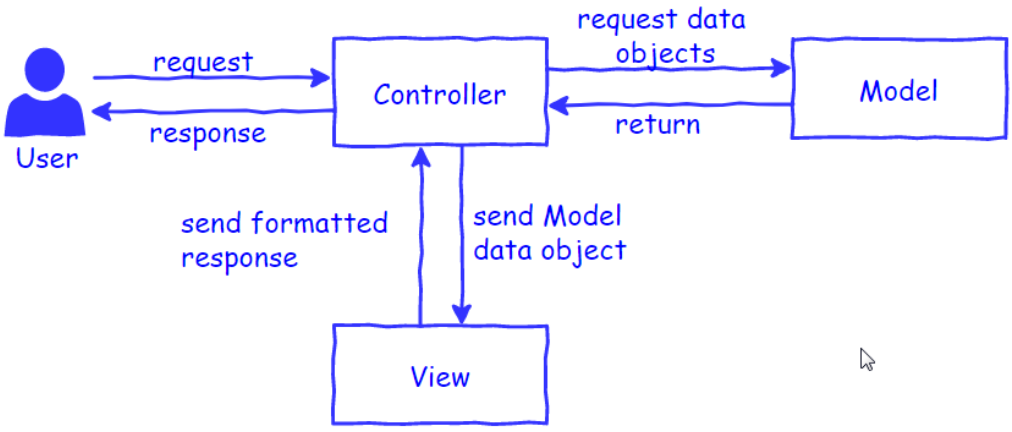
```
    }

  // Tạo mảng sinh viên, mỗi sinh viên có cấu trúc studentID => array('first_name' => 'First Name', 'last_name' => 'Last Name', 'grade' => XX.X)
    $students = array(
        256 => array('first_name' => 'Tuấn', 'last_name' => 'Trần Đăng', 'grade' => 98.5),
    2  => array('first_name' => 'An', 'last_name' => 'Nguyễn Xuân', 'grade' => 85.1),
    9  => array('first_name' => 'Dương', 'last_name' => 'Nguyễn Ngọc', 'grade' => 94.0),
        364 => array('first_name' => 'Chiến', 'last_name' => 'Hoàng Văn', 'grade' => 85.1),
    68  => array('first_name' => 'Phương', 'last_name' => 'Trần Thanh', 'grade' => 74.6)
    );

    // Tạo đối tượng
    $list = new StudentsList($students);

  // Hiển thị mảng trước khi sắp xếp
  echo '<h2>Danh sách gốc</h2>';
    $list->display();

  // Sắp xếp theo tên
    $list->sort(new MultiAlphaSort('first_name'));
  echo '<h2>Danh sách sắp xếp theo tên</h2>';
    $list->display();

    // Sắp xếp theo điểm
    $list->sort(new MultiNumberSort('grade', 'descending'));
    echo '<h2>Danh sách sắp xếp theo điểm</h2>';
    $list->display();

    // Xóa đối tượng
    unset($list);
    ?>
</body>
</html>
```

Trong ví dụ này chúng ta thấy rằng cùng là hành vi sort() nhưng nó khác nhau ở các thời điểm khác nhau, chỗ thì sắp xếp tên, chỗ thì sắp xếp điểm. Như vậy việc áp dụng Strategy Pattern giúp chúng ta thay đổi được hành vi của một đối tượng trong thời gian chạy. Kết quả khi thực hiện trong trình duyệt như sau:



5. Model - View - Controller MVC

Tần suất sử dụng 5/5, MVC pattern được dùng thường xuyên trong các framework hiện nay. MVC là một pattern dạng Architectural Design Pattern áp dụng khi xử lý các vấn đề liên quan đến kiến trúc ứng dụng.



Model là nơi lưu trữ dữ liệu người dùng, nó cho phép truy xuất dữ liệu để hiển thị hoặc thu thập dữ liệu. Model là cầu nối giữa thành phần View và Controller trong mẫu thiết kế này. Mục đích quan trọng nhất của nó là kết nối cơ sở dữ liệu, xử lý dữ liệu và chuẩn bị dữ liệu để chuyển đến các thành phần khác.

View là nơi dữ liệu được hiển thị, trong ứng dụng web View là một phần của hệ thống, nơi mà các mã HTML được sinh ra và hiển thị. View cũng là nơi nhận tương tác trực tiếp từ người dùng. Một vấn đề quan trọng là View không được lấy dữ liệu trực tiếp từ Controller mà phải thông qua Model.

Controller quản lý dữ liệu người dùng nhập vào và cập nhật sang Model, Controller chỉ được sử dụng khi có tương tác của người dùng, còn không nó không có giá trị. Controller chỉ đơn giản là thu thập thông tin và sau đó chuyển dữ liệu sang Model, nó không chứa bất kỳ logic nghiệp vụ nào. Controller kết nối với duy nhất một View và một Model tạo thành hệ thống dự liệu chạy theo một chiều (one way data flow system).

Chúng ta cùng đến với một ví dụ cụ thể để thấy được cách áp dụng MVC trong thực tế, trong ví dụ này chúng ta sẽ xây dựng một ứng dụng quản lý game thủ đế chế (Nhân sự kiện giải AOE Việt Trung 2017). Tạo ra thư mục OOP/MVC để bắt đầu bạn nhé. Đầu tiên chúng ta tạo ra file Model.php với nội dung:

```
<?php
class Model {
    private $gamers;

    public function __construct() {
        $this->gamers[] = array("name" => "Chim sẻ đi nắng",
"adress" => "Đan Phượng, Hà Tây", "city" => "Hà Nội");
    $this->gamers[] = array("name" => "Hồng Anh", "adress" => "Việt Trì, Phú Thọ", "city" =>
"Phú Thọ");
    $this->gamers[] = array("name" => "Gunny", "adress" => "Nhổn, Hà Nội", "city" => "Hà Nội");
    $this->gamers[] = array("name" => "Bibi", "adress" => "Hà Nội", "city" => "Hà Nội");
        $this->gamers[] = array("name" => "Văn Sự", "adress" =>
"Hoằng Hóa, Thanh Hóa", "city" => "Thanh Hóa");
    }

    public function getAllGamers() {
        return $this->gamers;
    }

    public function getGamerByName($gamerName) {
        $key = array_search($gamerName, array_column($this->gamers,
'name'));
        return $this->gamers[$key];
    }

    public function getGamersByCity($cityName) {
        $keys = array_keys(array_column($this->gamers, 'city'),
$cityName);
        $cityarray = array();
        foreach ($keys as $value) {
            $cityarray[] = $this->gamers[$value];
        }
        return $cityarray;
    }
```

```
    public function addGamer($name, $adress, $city) {
        $this->gamers[] = array("name" => $name, "adress" =>
$adress, "city" => $city);
    }
}
```

Tiếp theo chúng ta tạo ra View.php với nội dung:

```
<!DOCTYPE html>
<html lang="en">
    <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
  <title>Quản lý game thủ giải AOE Việt Trung 2017</title>

    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.
min.css" integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">
    <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements
and media queries -->
    <!-- WARNING: Respond.js doesn't work if you view the page via
file:// -->
    <!--[if lt IE 9]>
      <script
src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></scri
pt>
      <script
src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
    </head>
    <body>
        <div class="container">
     <h2><span class="glyphicon glyphicon-user"></span>Danh sách game thủ</h2>
            <div class="row">
                <div class="col-md-3">
                <ul>
                    <?php
                    foreach ($gamers as $g) {
                        echo '<li>';
                        echo '<a href="', $_SERVER['SCRIPT_NAME'],
'/PageController/getGamerByName/', $g['name'], '">', $g['name'],
'</a><br>';
                        echo $g['adress'], '<br>';
                        echo '--<a href="', $_SERVER['SCRIPT_NAME'],
'/PageController/getGamersByCity/', $g['city'], '">', $g['city'],
'</a>';
                        echo '</li>';
                    }
                    ?>
```

```
                </ul>
                </div><!--end col-->
                <div class="col-md-5">
                <ul>
                    <?php
                    if (isset($gamer)) {
            echo 'Thông tin chi tiết về game thủ:';
                        echo '<li>';
                        echo '<a href="', $_SERVER['SCRIPT_NAME'],
'/PageController/getGamerByName/', $gamer['name'], '">',
$gamer['name'], '</a><br></li>';
                        echo '<li>', $gamer['adress'], '</li>';
                        echo '<li>', $gamer['city'], '</li>';
                    }
                    if (isset($gamersInCities)) {
            echo 'Các game thủ ở: <b>', $gamersInCities[0]['city'], '</b>';
                        foreach ($gamersInCities as $g) {
                            echo '<li>';
                            echo '<a href="',
$_SERVER['SCRIPT_NAME'], '/PageController/getGamersByCity/',
$g['name'], '">', $g['name'], ' </a>';
                            echo $g['adress'], ' ';
                            echo $g['city'];
                            echo '</li>';
                        }
                    }
                    ?>
                </ul>
                </div><!--end col-->
            </div><!--end row-->
        </div>
    </body>
</html>
```

Cuối cùng chúng ta tạo ra Controller.php:

```
<?php
include_once './Model.php';

class Controller {
    private $data;
    private $model;

    public function handle() {
        $this->model = new Model();

        // Luôn lấy dữ liệu tất cả các game thủ và đưa vào thuộc
tính data
        $this->assign("gamers", $this->model->getAllGamers());

        // Tùy thuộc vào đường dẫn lấy các danh sách game thủ theo
bộ lọc
        if (isset($_SERVER['PATH_INFO'])) {
```

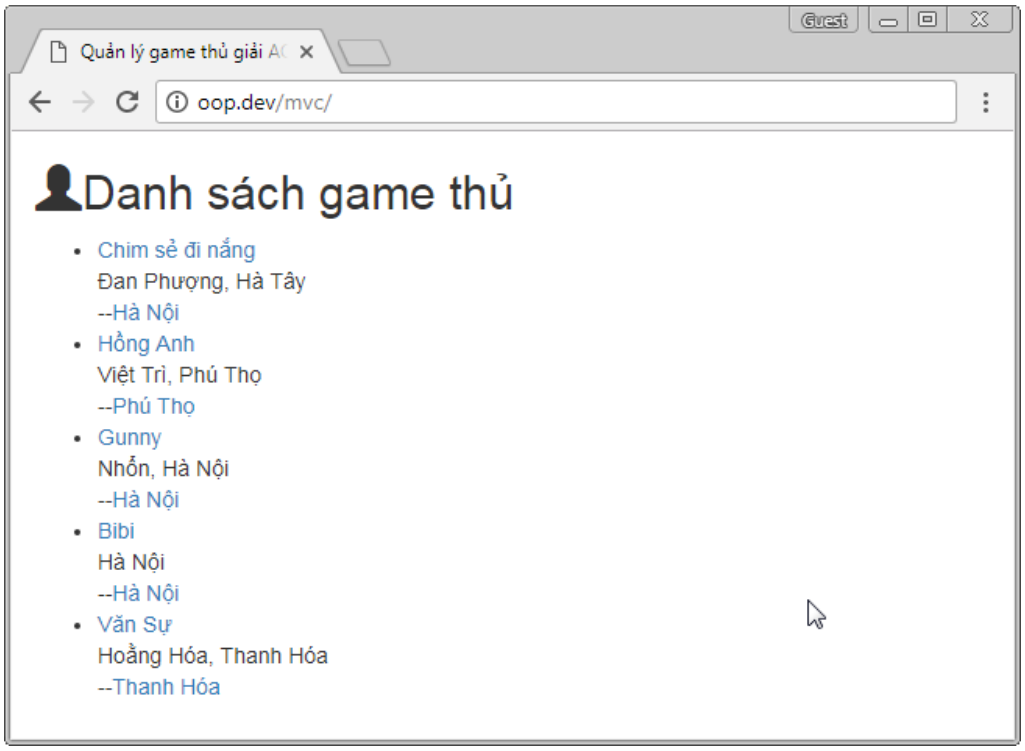
```
            // Ví dụ: index.php/PageController/getPersonByName/Chim
sẻ đi nắng
        // Convert chuỗi ký tự thành mảng, giá trị thứ 2 của mảng chứa action, giá trị thứ 3 chứa thông
tin của action
            $pathinfo = explode("/", $_SERVER['PATH_INFO']);

            switch ($pathinfo[2]) {
                case "getGamerByName";
                    // $arr[3]="Chim sẻ đi nắng"
                    $this-
>getGamerByName(urldecode(trim($pathinfo[3])));
                    break;
                case "getGamersByCity":
          // $arr[3]="Hà Nội"
                    $this-
>getGamersByCity(urldecode(trim($pathinfo[3])));
                    break;
                default:
                    $this->display("./View.php");
            }
        } else {
            $this->display("./View.php");
        }
    }

    public function getGamerByName($name) {
        $this->assign("gamer", $this->model->getGamerByName($name));
        $this->display("./View.php");
    }

    public function getGamersByCity($city) {
        $this->assign("gamersInCities", $this->model-
>getGamersByCity($city));
        $this->display("./View.php");
    }

    private function assign($key,$value){
        $this->data[$key]=$value;
    }

    private function display($htmlPage){
        extract($this->data);
        include_once $htmlPage;
    }

}
```

Như vậy, chúng ta đã có đủ 3 thành phần Model, View và Controller, tiếp theo chúng ta sử dụng cả 3 thành phần này để xử lý ứng dụng. Tạo file index.php là nơi tiếp nhận yêu cầu từ người dùng:

```
?php
include('./Controller.php');
```

```
$controller = new Controller();
$controller->handle();
```

File này chỉ đơn giản là tạo ra một instance từ lớp Controller. Bạn có thể xem lại sơ đồ MVC ở đầu phần này, đây chính là nơi người dùng gửi yêu cầu HTTP đến hệ thống. Tiếp theo, Controller sẽ thực hiện gửi yêu cầu (gọi phương thức lấy dữ liệu từ Model) đến Model và Model trả về các dữ liệu cần thiết, các yêu cầu này phụ thuộc vào nhập liệu của người dùng là tham số thứ 2 trong đường dẫn, từ đó Controller có thể gọi đến các phương thức khác nhau của Model như getAllGamers, getGamerByName, getGamersByCity. Sau khi có dữ liệu trả về Controller gửi dữ liệu đến View thông qua việc gọi $this->display("./View.php"); và View trả về định dạng HTML để hiển thị cho người dùng. Chúng ta cùng truy cập vào http://oop.dev/mvc để xem kết quả:



Khi click vào tên từng game thủ chúng ta có chi tiết thông tin game thủ đó, khi click vào từng thành phố sẽ ra danh sách các game thủ trong thành phố.

6. Lời kết

Design Pattern được sử dụng để giải quyết các vấn đề chung, nó không phải được áp dụng để làm phức tạp ứng dụng, do vậy việc hiểu rõ Design Pattern là gì và cách sử dụng Design Pattern như thế nào? là rất quan trọng. Design Pattern là một con dao hai lưỡi, nó có thể giải quyết những vấn đề phức tạp nhưng cũng có thể làm phức tạp một vấn đề đơn giản. Trong một bài viết ngắn không thể tường tận mọi vấn đề của thiết kế mẫu trong lập trình, chúng ta sẽ còn bàn luận lại vấn đề này một cách chi tiết hoặc ở một góc nhìn khác, bạn nhớ đón đọc nhé.

Bài 5. Design Patterns: Best Practices for Application Development

Cho dù bạn đang xây dựng một trang web, ứng dụng web hay ứng dụng dành cho thiết bị di động, sự thành công của dự án sẽ phụ thuộc vào mức độ nhà phát triển có thể đáp ứng các yêu cầu của dự án. Đối với các dự án lớn hơn, có thể có một nhóm các nhà phát triển phải làm việc cùng nhau để đạt được mục tiêu của họ.

Phát triển trong một môi trường phức tạp là một trong những lý do chính khiến khái niệm “design pattern” ban đầu được giới thiệu. Design pattern là giải pháp cho các vấn đề thiết kế phần mềm xảy ra nhất quán trong quá trình phát triển ứng dụng trong thế giới thực. Những patten này trình bày một khuôn khổ có thể tái sử dụng của các thiết kế và tương tác của các đối tượng.

Một trong những khía cạnh mà tôi thích về các mẫu thiết kế là để một nhà phát triển thậm chí nắm bắt được giá trị của việc sử dụng một cấu trúc như vậy, họ sẽ phải đạt được một số mức độ tinh vi với cách họ nghĩ về mã. Áp dụng các mẫu thiết kế trong các dự án phần mềm có nghĩa là cả hai đều quan tâm đến việc xây dựng một mối quan

hệ OOP tối ưu để giải quyết vấn đề một cách duyên dáng, cũng như chống lại sự cám dỗ của chính họ trong việc viết mã thứ gì đó theo ý họ.

Các mẫu thiết kế cung cấp một nền tảng chung và một thuật ngữ tiêu chuẩn cho các nhà phát triển. Khi các nhà phát triển đang sử dụng một mẫu thiết kế singleton, đồng đội của họ sẽ hiểu rằng một đối tượng duy nhất đang được sử dụng.

Khi các mẫu thiết kế đã phát triển trong hơn 20 năm qua, chúng đã trở thành các phương pháp hay nhất cho các vấn đề phát triển chung. Nhận thức được các mẫu thiết kế cũng có thể giới thiệu các phương pháp hay nhất này và hỗ trợ các nhà phát triển ít kinh nghiệm hơn học thiết kế phần mềm theo cách hợp tác và hiệu quả hơn.

Khái niệm về các mẫu thiết kế trong phát triển phần mềm ban đầu được trình bày trong cuốn sách Năm 1994 "***Design Patterns: Elements of Reusable Object-Oriented Software***". Cuốn sách được viết bởi bốn tác giả, những người hiện được gọi chung là "Gang of Four".

Được coi là một nguồn quan trọng cho lý thuyết và thực hành thiết kế hướng đối tượng, cuốn sách đã trở nên có ảnh hưởng lớn trong lĩnh vực kỹ thuật phần mềm. Hai chương đầu tiên của Design Patterns tập trung vào khám phá các khả năng và cạm bẫy của lập trình hướng đối tượng. Phần còn lại của cuốn sách mô tả 23 mẫu thiết kế phần mềm.

23 mẫu do Gang of Four phát triển thường được coi là nền tảng cho tất cả các mẫu khác. Chúng được phân loại thành ba nhóm: **Creational**, **Structural**, **Behavioral**

Nguồn gốc của các mẫu thiết kế cho chúng ta biết rất nhiều điều về lý do tại sao chúng tồn tại. Vào những năm 1970, ngoài những ngày Xerox PARC nổi tiếng, đã phát sinh ra một ngôn ngữ lập trình hướng đối tượng có tên là Smalltalk, ngôn ngữ này rất có ảnh hưởng đến tất cả các thiết kế của ngôn ngữ lập trình sau này.

Mức độ trừu tượng hiện có thể cung cấp khả năng tạo ra các mối quan hệ khác nhau giữa các đối tượng. Vào những năm 1990 khi các ngôn ngữ lập trình hướng đối tượng được phổ biến rộng rãi, có nhiều cách bạn có thể xây dựng các đối tượng để thực hiện những việc hoàn thành một nhiệm vụ, với rất ít sự đồng thuận về cách nào là tốt nhất.

Sau đó, rõ ràng là nhiều vấn đề thiết kế phần mềm lặp đi lặp lại các vấn đề giống nhau, và các nhà phát triển không cần phải phát minh lại bánh xe mỗi lần. Ngoài ra còn có 23 vấn đề và giải pháp hướng đối tượng phổ biến của Gang of Four hiện được gọi chung là các mẫu thiết kế.

Tôi nghĩ rõ ràng là tại sao chúng ta nên sử dụng các mẫu thiết kế, cũng vì lý do đó mà chúng ta nên "đứng trên vai những người khổng lồ", tức là tận dụng những phát minh / giải pháp tuyệt vời mà những người khác đã sản xuất trước chúng ta. Nhưng đây là một khía cạnh vô hình không thể nhìn thấy rõ ràng, nó không phải là một khách hàng phải đối mặt với việc có thể giao hoặc thậm chí là một nhiệm vụ trong một dự án phát triển, mà đó là ứng dụng chính xác của việc giải quyết các vấn đề trong nghệ thuật lập trình máy tính.

Tuy nhiên, việc thiếu áp dụng các mô hình thành công chắc chắn sẽ có thể nhìn thấy được do ảnh hưởng của việc tăng thời gian / nỗ lực phát triển / chi phí / lỗi / làm lại và không có khả năng dễ dàng thích ứng với các yêu cầu thay đổi của khách hàng.

Các mẫu thiết kế là một công cụ tuyệt vời không chỉ giúp các nhà phát triển làm việc trong một khuôn khổ gắn kết mà còn là một nguồn tài nguyên tuyệt vời để xác định người viết mã cho nhóm của bạn.

23 Gang of Four Design Patterns:

Creational Patterns	
Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist
Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object's interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Facade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object
Behavioral Patterns	

Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

Bài 6. Vỡ lòng về bộ nguyên tắc thiết kế SOLID

Kiến trúc

Hôm nay phần mềm của bạn vẫn phục vụ tốt với bộ tính năng mà nó cung cấp, ngày mai người dùng có thể sẽ yêu cầu bạn sản xuất tính năng mới, và họ sẽ luôn làm thế. Bạn đang sản xuất dở phần mềm, hôm nay bạn có 50% tính năng và ngày mai sẽ làm 51%. Chúng ta gọi công cuộc thêm tính năng mới này là "mở rộng" phần mềm.

Khi viết phần mềm, chúng ta luôn phải quan tâm đến việc duy trì *khả năng mở rộng* của phần mềm. Dân sản xuất phần mềm truyền nhau hài ngôn sau để nói về việc mất khả năng mở rộng:

Chúng ta sẽ dành 10% thời gian đầu tiên của dự án để phát triển 90% tính năng, và chúng ta sẽ dùng 90% thời gian sau đó để phát triển 10% tính năng cuối cùng (mà không biết có xong không).

Khả năng mở rộng phần mềm cũng giống như khả năng xây thêm tầng cho một ngôi nhà. Khả năng đó trước hết đến từ hai thứ: vật liệu tốt, kiến trúc tốt, và nền móng tốt. Đối ứng với phần mềm thì vật liệu tốt chính là mã sạch và kiến trúc tốt là thiết kế.

Nguyên tắc thiết kế

Mã sạch cũng giống như gạch tốt. Dĩ nhiên gạch tốt là một trong những điều đầu tiên chúng ta cần quan tâm đến khi xây dựng một ngôi nhà. Nhưng để ngôi nhà được vững chắc và duy trì được khả năng sử dụng cao khi xây dựng thêm các tầng mới thì chỉ gạch tốt là chưa đủ. Chúng ta cần đến thiết kế. Và như thế chúng ta tìm đến các nguyên tắc SOLID.

Nếu như các nguyên tắc *Mã sạch* hướng dẫn chúng ta viết nên các hàm và các class tốt, các nguyên tắc SOLID chỉ ra cách đặt các hàm và cấu trúc dữ liệu và các class, cũng như cách các class nội kết với nhau. Dụng ngôn *"class"* không có nghĩa rằng SOLID chỉ áp dụng cho phần mềm hướng đối tượng. Class chỉ thuần túy là một tập các hàm và các dữ liệu được nhóm lại với nhau. Bất kỳ phần mềm nào cũng có những nhóm như thế, cho dù chúng có được gọi là class hay không. Và SOLID nhắm đến những nhóm này.

Mục tiêu của SOLID là giúp tạo ra những cấu trúc phần mềm cấp trung mang tính uyển chuyển, dễ hiểu, và có khả năng dùng làm cơ sở để tạo thành những component có thể dùng chung cho nhiều hệ thống phần mềm. "Cấp trung" nói lên thực tế rằng chúng ta dùng đến SOLID khi tập trung vào cách kết cấu các khối và cấu phần trong phần mềm, thay vì đi vào chi tiết mã lệnh.

SOLID

Các nguyên tắc SOLID không được hình thành ngay một lúc mà trải qua một lịch sử dài. Trong quá trình tìm kiếm các nguyên tắc cốt lõi khi thiết kế các cấu trúc cấp trung, các thợ cả trong nghành thủ công phần mềm đã tạo ra nhiều bộ nguyên tắc khác nhau. Trong số có tới nay còn lại năm nguyên tắc dễ hiểu, thanh nhã, và đủ tốt; mà nếu sắp xếp theo đúng thứ tự thì chúng ta sẽ có tính từ *solid* (rắn chắc).

Sơ bộ, SOLID gồm 5 nguyên tắc như sau:

- ***Single Responsibility Principle*** – nguyên tắc này yêu cầu mỗi khối phần mềm chỉ có một lý do để thay đổi.
- ***Open-Closed Principle*** – phần mềm luôn phải ở trạng thái sao cho có thể thay đổi hành vi của chúng bằng cách thêm mã mới thay vì sửa mã cũ.
- ***Liskov Substitution Principle*** – phần mềm luôn phải ở trạng thái mà mỗi thành phần đều có thể thay thế mà không ảnh hưởng đến hành vi của nó.

- ***Interface Segregation Principle*** – tránh tạo ra quan hệ phụ thuộc với những thứ không dùng đến
- ***Dependency Inversion Principle*** – mã triển khai chính sách cấp cao không được phụ thuộc vào triển khi chi tiết ở mức thấp. Thay vì thế chi tiết nên phụ thuộc vào chính sách.

Các nguyên tắc này đã được mô tả trong rất nhiều ấn phẩm trong suốt nhiều năm. Bài viết này sẽ đi vào ý nghĩa thiết kế đằng sau chúng, thay vì tiếp tục các tranh luận chi tiết trong khái niệm.

Nguyên tắc Đơn Trách Nhiệm

Đơn Trách Nhiệm, hay *Trách Nhiệm Duy Nhất*, có lẽ là nguyên tắc được nghe đến nhiều nhất và bị hiểu sai nhiều nhất trong số các nguyên tắc SOLID. Cứ mười anh lập trình viên thì phải đến hơn chín anh cho rằng nguyên tắc này phát biểu điều gì đó liên quan đến mỗi hàm (hay tệ hơn – mỗi class???) chỉ được làm một việc.

Thật ra không thể trách được, quả thật là có một nguyên tắc như thế. Một hàm chỉ được phép làm một và chỉ một, việc. Chúng ta áp dụng nguyên tắc đó khi thực hiện tái cấu trúc những hàm lớn thành những hàm nhỏ hơn; chúng ta áp dụng nguyên tắc đó lại mức thấp của công việc viết mã. Nhưng đó không phải là một trong các nguyên tắc SOLID, càng không phải Nguyên Tắc Đơn Trách nhiệm.

Trong lịch sử, *Nguyên Tắc Đơn Trách Nhiệm* từng được phát biểu như sau:

Một module chỉ được có một và chỉ một lý do để thay đổi.

Các sản phẩm phần mềm thay đổi là để đáp ứng người dùng và các bên liên quan. Họ là các *lý do để thay đổi* mà nguyên tắc nói tới. Thế nên *Đơn Trách Nhiệm* có thể được phát biểu lại như sau:

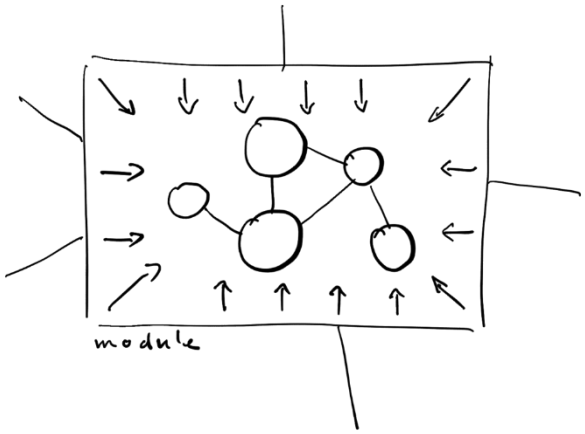
Một module chỉ phải chịu trách nhiệm thay đổi trước một và chỉ một người dùng hay bên liên quan.

Những từ "người dùng" và "bên liên quan" không thực sự đúng cho lắm. Đôi khi có nhiều hơn một người dùng hay bên liên quan muốn hệ thống thay đổi nhưng theo cùng một cách. Cái chúng ta thực sự muốn nói đến là một nhóm — một hay nhiều người ra yêu cầu thay đổi. Chúng ta gọi nhóm đó là một *tác nhân*. Theo đó, định nghĩa cuối cùng của *Nguyên Tắc Đơn Trách Nhiệm* sẽ là:

Một mô-đun chỉ phải chịu trách nhiệm trước một, và chỉ một, tác nhân.

Tiếp theo là đến ý nghĩa của khái niệm *mô-đun*. Định nghĩa đơn giản nhất của mô-đun là một tập tin mã nguồn. Đa phần chúng ta có thể dùng định nghĩa đó. Có vài ngôn ngữ và môi trường phát triển đặc thù không sử dụng tập tin để chứa mã nguồn của chúng. Tuy nhiên dù trong bất kỳ môi trường nào cũng luôn tồn tại các nhóm cấu trúc dữ liệu và hàm tụ về một hướng, mà chúng ta gọi là sự ngưng tụ, sự ngưng tụ cố kết

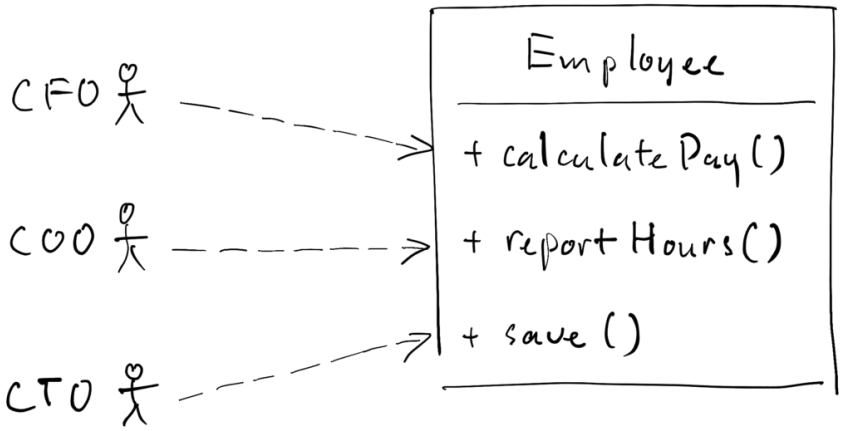
các thành phần rời rạc lại với nhau và tạo thành mô-đun. Nói tóm lại, mô-đun là *một tập hợp ngưng tụ các hàm và cấu trúc dữ liệu*.



Giờ chúng ta sẽ làm sáng tỏ *Nguyên Tắc Đơn Trách Nhiệm* thông qua một vài ví dụ về sự vi phạm nó.

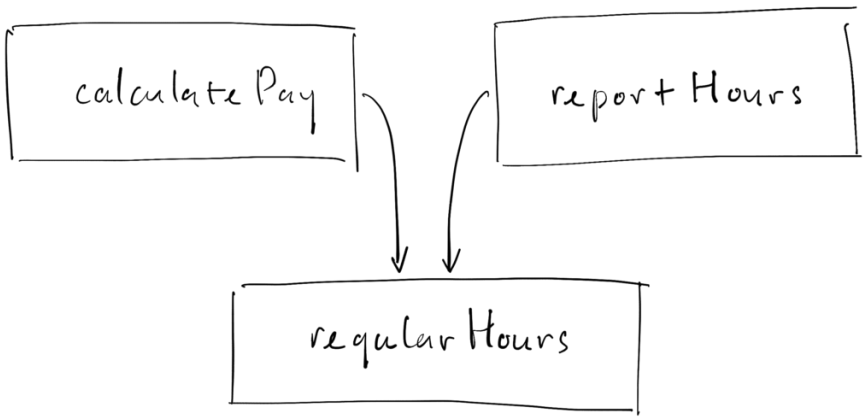
Triệu chứng: xung đột nghiệp vụ

Trong ví dụ dưới đây có lớp Employee của một chương trình tính lương. Nó có các phương thức calculatePay()được sử dụng bởi phòng kế toán để tính lương, phục vụ giám đốc tài chính; reportHours() được sử dụng bởi phòng nhân sự để tính ngày công, phục vụ giám đốc nhân sự; và save() được sử dụng bởi các quản trị viên cơ sở dữ liệu để lưu thông tin nhân viên, phục vụ giám đốc công nghệ. Các giám đốc rõ là các tác nhân khác nhau, và lớp này đã vi phạm *Nguyên Tắc Đơn Trách Nhiệm*.



Bằng việc ngưng tụ cả ba phương thức vào cùng một lớp, các nhà phát triển đã ràng buộc ba tác nhân khác nhau với nhau. Một yêu cầu thay đổi nào đó từ phía COO sẽ có thể gây ảnh hưởng tới nghiệp vụ của CTO.

Lấy ví dụ, calculatePay() và reportHours sử dụng cùng một công thức để quy số giờ làm việc thành giờ làm việc tiêu chuẩn (những giờ làm việc quá giờ sẽ có hệ số cao hơn 1 chẳng hạn). Và bởi khử mã lặp nên các nhà phát triển đã đặt công thức này vào một hàm dùng chung tên là regularHous():



Rồi một ngày, CFO quyết định rằng công thức cần phải thay đổi một chút, nhưng đội ngũ của COO thì không có nhu cầu với thay đổi này, do họ dùng con số giờ làm việc tiêu chuẩn cho những mục đích khác nhau.

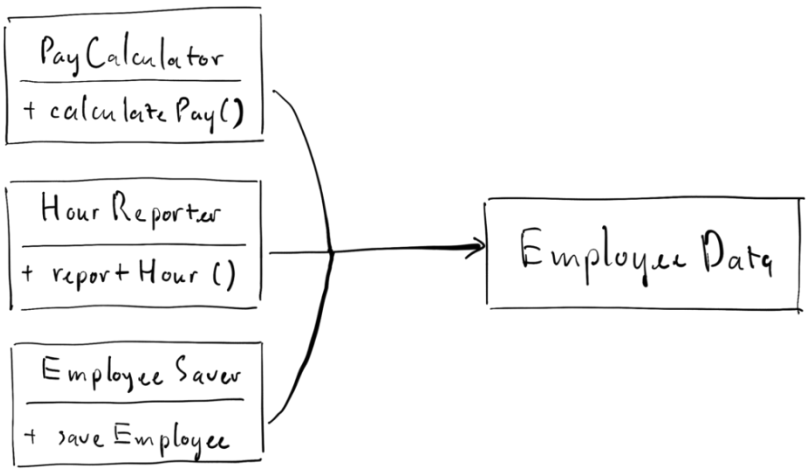
Người lập trình viên được giao nhiệm vụ triển khai thay đổi không nhận ra rằng regularHours cũng được sử dụng bởi reportHours, anh ta đã thực hiện sửa đổi, kiểm thử cẩn thận, đội ngũ của CFO đã kiểm tra, chức năng hoạt động như mong muốn, được nghiệm thu và được đưa vào thực tế.

Đội ngũ của COO phải rất lâu sau mới nhận ra rằng những con số mà họ dựa vào để báo cáo và ra quyết định đang có vấn đề. Và tới khi đó thì hậu quả đã là rất nhiều tài nguyên và nỗ lực đã bị lãng phí.

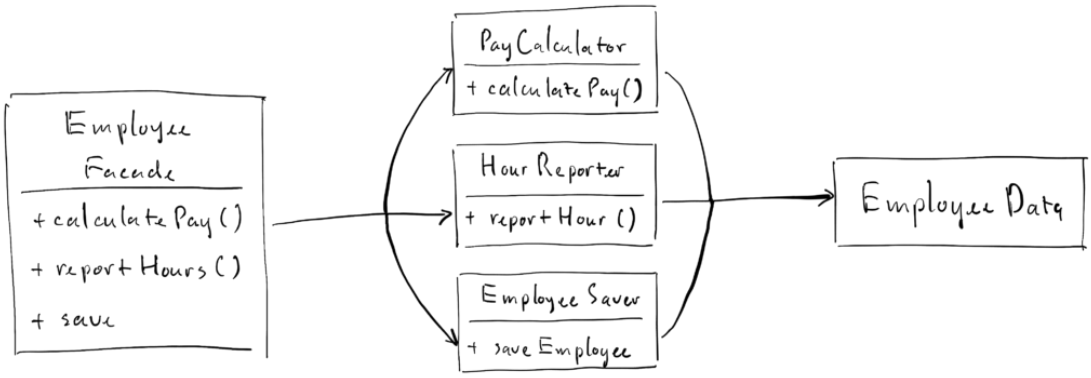
Bất kỳ ai có thâmtrong nghành cũng đều đã thấy những chuyện tương tự. Chúng xảy ra bởi vì chúng ta đã đặt những mã nguồn phụ thuộc vào những tác nhân khác nhau lại gần với nhau quá. *Nguyên Tắc Đơn Trách Nhiệm* dặn chúng ta đặt chúng xa ra.

Giải pháp

Có nhiều giải pháp khác nhau cho vấn đề này. Mỗi giải pháp lại đặt các hàm vào các lớp khác nhau. Có lẽ cách dễ nhận thấy nhất đó là tách dữ liệu khỏi các hàm, và đặt các hàm chức năng vào trong những lớp chỉ chứa đủ mã nguồn cần thiết để chức đó hoạt động. Các lớp mang chức năng không biết về sự tồn tại của nhau, do đó tránh được bất kỳ sự xung đột nghiệp vụ nào.

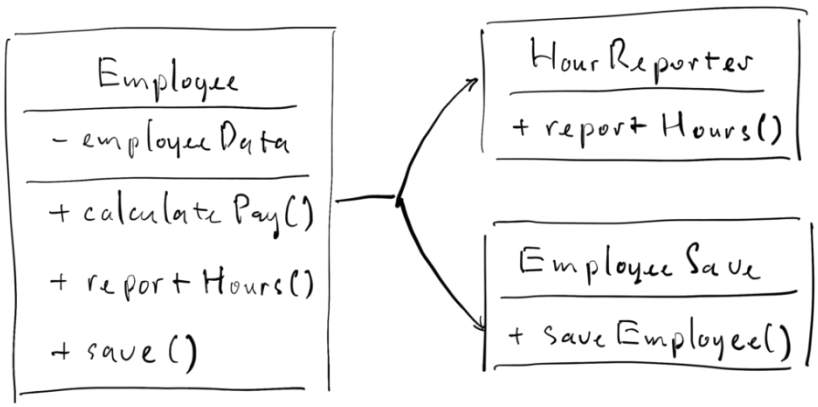


Nhược điểm của giải pháp này chính là việc lập trình viên sẽ phải quan tâm đến những ba lớp khác nhau. Cách giải quyết là sử dụng mẫu thiết kế *Facade*:



Lớp EmployeeFacade chứa rất ít mã. Nó chỉ chịu trách nhiệm khởi tạo và ủy thác công việc cho các lớp mang chức năng.

Một số nhà phát triển thích giữ những nghiệp vụ quan trọng nhất được gần với dữ liệu hơn. Điều này có thể được thực hiện bằng cách giữ phương thức quan trọng nhất trong lớp Employee ban đầu, và dùng lớp này làm *facade* cho các hàm chức năng nhỏ hơn.



Bạn có thể sẽ cảm thấy muốn chối bỏ các giải pháp trên bởi việc mỗi lớp chỉ chứa một hàm trông không được tự nhiên lắm. Thực tế điều này hiếm khi xảy ra. Mỗi lớp luôn chứa cả các hàm riêng tư, chẳng hạn để phục vụ cho chức năng tính lương, tính giờ làm hay lưu tồn dữ liệu, và số lượng của chúng thường không ít.

Nguyên Tắc Đóng/Mở

Nguyên Tắc Đóng/Mở được phát biểu như sau:

Một tạo tác phần mềm nên mở cửa cho sự mở rộng nhưng đóng lại với những sửa đổi

Nói cách khác, *một tạo phẩm phần mềm phải hành xử theo lối có khả năng mở rộng mà không phải sửa đổi tạo phẩm đó*. Đây, cơ bản là lý do khiến chúng ta nghiên cứu về kiến trúc phần mềm. Nếu chỉ một vài mở rộng đơn giản theo yêu cầu cũng kéo theo những thay đổi lớn trên phần mềm, thì rõ ràng các kiến trúc sư của hệ thống phần mềm đó đã trên bờ vực mất kiểm soát kiến trúc đến nơi.

Hầu hết người học thiết kế phần mềm đều coi *Nguyên Tắc Đóng/Mở* như một nguyên tắc hướng dẫn khi thiết kế các lớp và mô-đun. Nhưng không chỉ thế, nguyên tắc này còn phát huy ý nghĩa của nó khi chúng ta thiết kế các cấu phần của hệ thống phần mềm.

Hãy xem xét ví dụ sau.

Ví dụ: hệ thống báo cáo tài chính

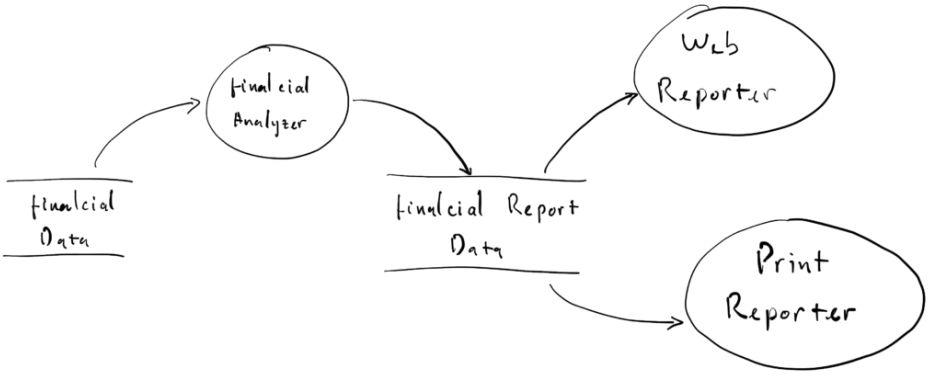
Hãy tưởng tượng chúng ta có một hệ thống hiển thị thông tin tài chính tổng hợp lên trang web. Danh mục thông tin rất dài, và có thể cuộn để xem. Các số âm được hiển thị bằng màu đỏ.

Bây giờ các bên liên quan yêu cầu một khối thông tin tương tự nhưng ở định dạng sẵn sàng để in lên máy in đen trắng. Danh mục được phân trang, và các số âm được bao quanh bởi cặp dấu ngoặc đơn.

Rõ ràng phải viết thêm mã mới, nhưng bao nhiêu mã cũ sẽ phải thay đổi? Hệ thống phần mềm được thiết kế tốt sẽ giảm số lượng mã cũ phải sửa xuống tối thiểu. Lý tưởng nhất là không có.

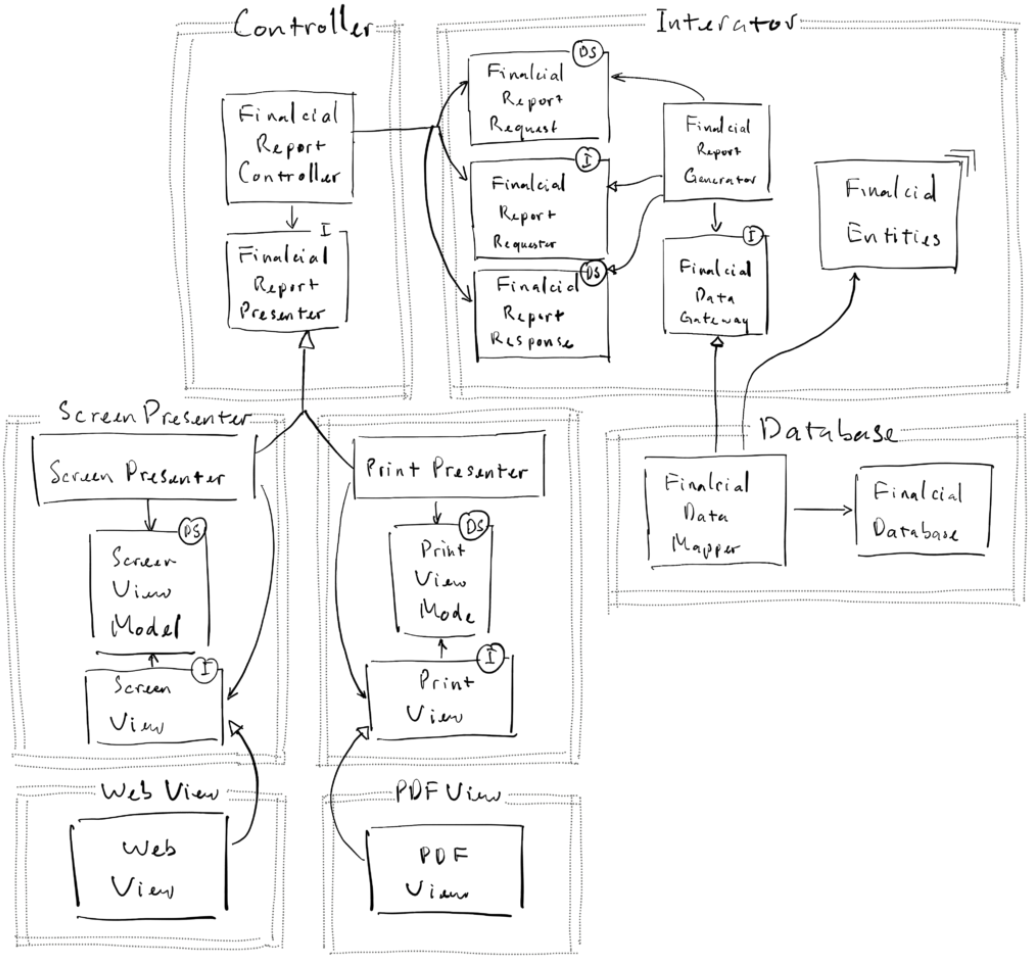
Làm thế nào? Bước đầu tiên là tổ chức phân tách những cấu phần chịu trách nhiệm bởi những tác nhân khác nhau, sau đó tổ chức các mối phụ thuộc giữa các cấu phần đó một cách hợp lý.

Với *Nguyên Tắc Đơn Trách Nhiệm*, chúng ta có được cái nhìn tổng thể về luồng dữ liệu như dưới đây. Thủ tục *Phân tích Tài chính* sẽ tạo ra *Dữ liệu Báo cáo*, thứ sau đó được định dạng cho phù hợp bởi hai *Trình tạo Báo cáo*.



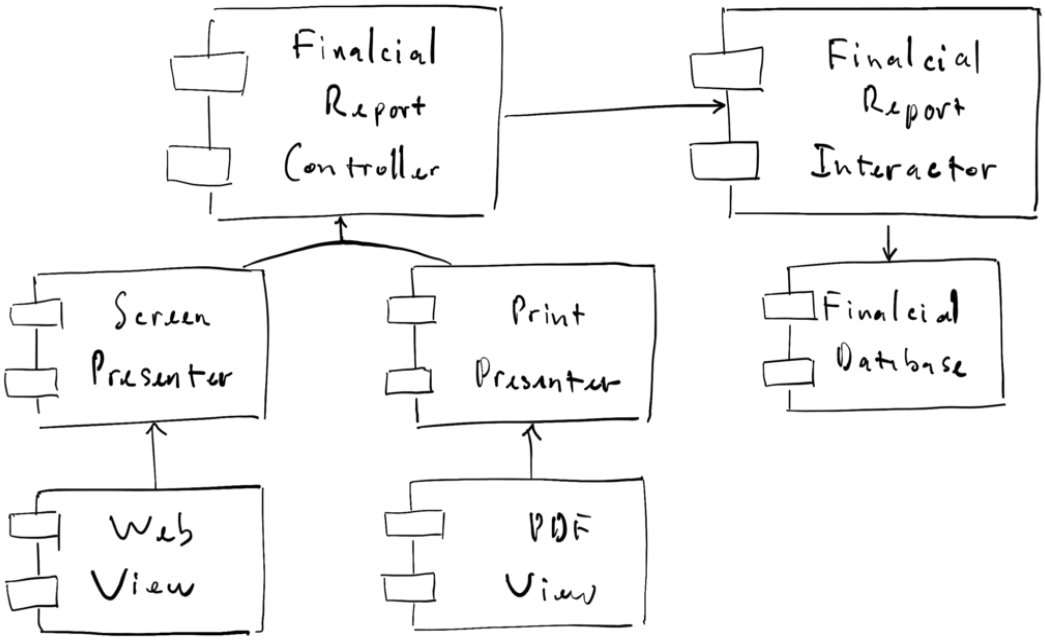
Bước tiếp theo là tổ chức các mối phụ thuộc trong mã nguồn sao cho các thay đổi từ một trong các trách nhiệm không kéo theo thay đổi trên các trách nhiệm còn lại. Đồng thời cũng phải đảm bảo rằng hành vi có thể được mở rộng mà không cần phải sửa đổi mã nguồn của hành vi cũ.

Chúng ta tổ chức các tiến trình vào các lớp và phân bổ các lớp đó vào các cấu phần, được thể hiện bằng các khu vực với đường bao kép, như trong hình dưới đây. Các cấu phần bao gồm *Controller*, *Interator*, *Database*, các *Presenter*, và các *View*.



Trong sơ đồ, các ký hiệu I là các *interface*, DS là các cấu trúc dữ liệu, các mối quan hệ *dùng đến* được thể hiện bởi các mũi tên hở, và các quan hệ *thi hành* hay *kế thừa* được thể hiện bởi các mũi tên kín.

Hãy để ý vào các mũi tên cắt ngang qua đường biên giữa các cấu phần, **chúng là những mũi tên đơn hướng**. Những mũi tên hướng về những cấu phần mà chúng ta muốn bảo vệ khỏi sửa đổi.



Nếu cấu phần A cần được bảo vệ khỏi những thay đổi từ cấu phần B, thì cấu phần B nên phụ thuộc vào cấu phần A.

Chúng ta muốn bảo vệ *Controller* khỏi những thay đổi của các *Presenter*. Chúng ta muốn bảo vệ các *Presenter* khỏi những thay đổi trong các *View*. Chúng ta muốn bảo vệ *Interactor* khỏi những thay đổi ở — bất cứ nơi nào khác. Các thay đổi trên *View, Presenter, Controller,* hay *Database* sẽ không làm thay đổi *Interactor*.

Tại sao *Interactor* lại được lưu tâm như vậy? *Interactor* chứa những quy tắc nghiệp vụ, chứa các chính sách ở cấp cao nhất của ứng dụng. *Interactor* là trung tâm và tất cả các cấu phần khác là thiết bị ngoại vi.

Controller là ngoại vi của Interactor, nhưng là trung tâm đối với *Presenter*; và tương tự như thế, *Presenter* là trung tâm của các *View*. Chúng ta gọi đây là hệ thống cấp bậc bảo vệ. *View* là cấu phần ở cấp thấp nhất, vì vậy chúng ít được bảo vệ nhất, *Presenter* có cấp cao hơn *View* nhưng thấp hơn so với *Controller* và *Interactor*.

Đó la sự hoạt động của *Nguyên Tắc Đóng/Mở* ở mức độ kiến trúc. Người kiến trúc sư thực hiện phân tách các chức năng bằng cách đặt ra những câu hỏi như thế nào, tại sao, bao giờ các thay đổi sẽ xuất hiện, sau đó tổ chức các chức năng riêng biệt vào một hệ thống phân cấp. Các cấu phần nằm ở mức cao hơn trong hệ thống được bảo vệ khỏi những thay đổi nằm ở cấu phần thấp hơn.

Hướng điều khiển

Đừng hoảng hốt trước mức độ chi tiết quá mức của thiết kế ở trên, hầu hết các sự phức tạp trong sơ đồ là để đảm bảo rằng các mối phụ thuộc băng qua đường biên giữa các cấu phần được chỉ theo đúng hướng.

Ví dụ: <I>FinancialDataGateway tồn tại giữa FinancialReportGenerator và FinancialDataMapper là để để đảo ngược sự phụ thuộc mà lẽ ra đã chỉ từ hướng *Interactor* đến *Database*. Tương tự với <I>FinancialReportPresenter và hai giao diện *View*.

Ẩn giấu thông tin

Giao diện <I>FinancialReportRequester phục vụ một cho mục đích khác. Nó ở đó để đảm bảo FinancialReportController không *bị* biết quá nhiều về cấu trúc bên trong của *Interactor*. Nếu không có nó, *Controller*sẽ có quá nhiều mối phụ thuộc vào *FinancialEntities*.

Các phụ thuộc bắc cầu vi phạm nguyên tắc chung rằng các **thực thể phần mềm không nên phụ thuộc vào những thứ chúng không trực tiếp sử dụng**. Chúng ta sẽ gặp lại nguyên tắc đó khi nói về *Nguyên Tắc Phân Tách Giao Diện*.

Vì vậy, tuy nói rằng ưu tiên hàng đầu của chúng ta là bảo vệ *Interactor* khỏi các thay đổi của *Controller*, nhưng chúng ta cũng muốn bảo vệ *Controller* khỏi các thay đổi của *Interactor* bằng cách ẩn đi cấu trúc nội bộ của *Interactor*.

Kết luận

Nguyên Tắc Đóng/Mở là một trong những lực lượng ngầm lèo lái kiến trúc của hệ thống phần mềm. Mục tiêu là làm cho hệ thống dễ dàng mở rộng mà không phải gây ra những thay đổi có tác động lớn. Mục tiêu này được thực hiện bằng cách quy hoạch hệ thống thành các cấu phần và phân bố các cấu phần đó vào một hệ thống có thứ bậc để bảo vệ các cấu phần cấp cao khỏi những thay đổi trong các cấu phần cấp thấp hơn.

Nguyên Tắc Thay Thế Liskov

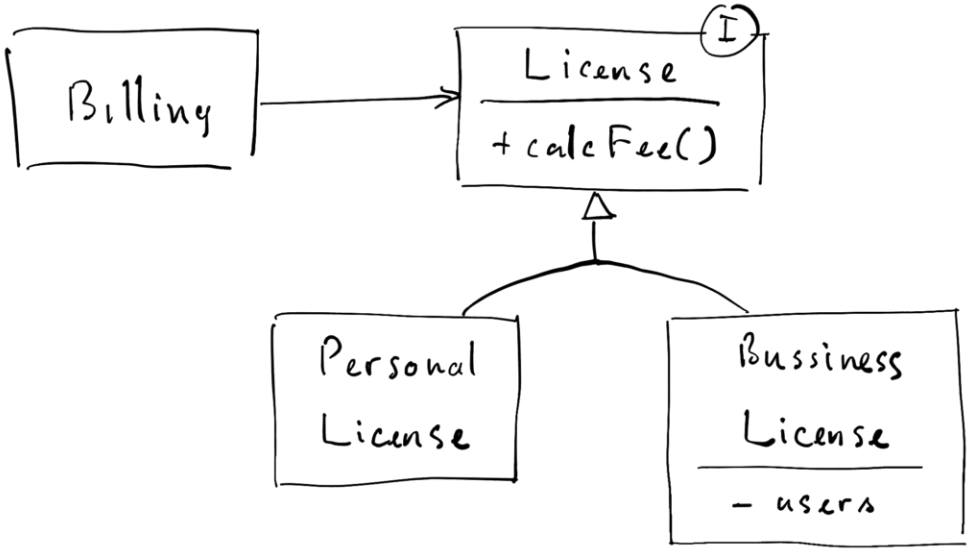
Năm 1988, Barbara Liskov, để định nghĩa một kiểu con, đã viết như sau, trong paper Thông báo SIGPLAN 23, 5 - Trừu tượng hóa và phân cấp dữ liệu (tháng 5 năm 1988).

Điều muốn có ở đây là một cái gì đó giống như thuộc tính thay thế như sau đây: Nếu với mỗi đối tượng **o1** có kiểu **S** tồn tại một đối tượng **o2** có kiểu **T** sao cho với tất cả các chương trình **P** được xác định theo **T** thì hành vi của **P** không thay đổi khi lấy **o1** thay thế cho **o2**, thì **S** là một kiểu con của **T**.

Ý tưởng này được gọi là *Nguyên Tắc Thay Thế Liskov*, để hiểu nó chúng ta hãy xem xét một số ví dụ.

Dẫn dắt quan hệ kế thừa

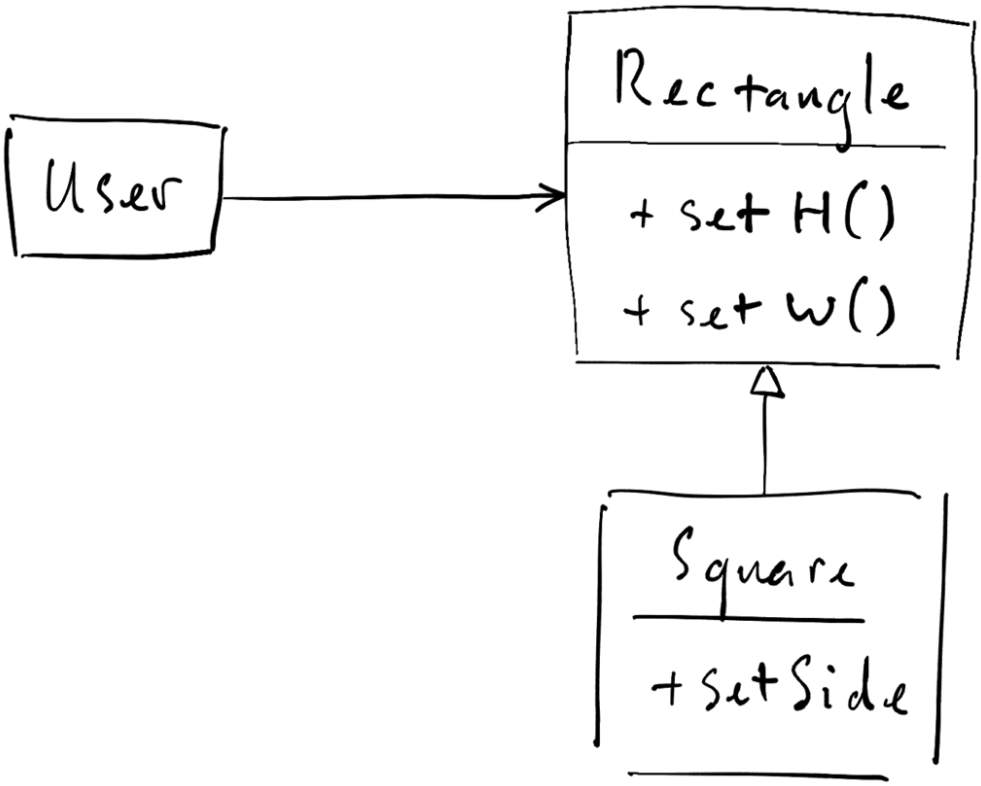
Tưởng tượng rằng chúng ta có một lớp có tên là License như trong hình dưới đây. Lớp này có một phương thức có tên là calcFee(), được gọi bởi ứng dụng *Billing*. License có hai lớp "con": PersonalLicense và BusinessLicense. Chúng dùng các thuật toán khác nhau để tính *phí*.



Thiết kế này phù hợp với *Nguyên Tắc Thay Thế Liskov* vì hành vi của ứng dụng *Billing* không phụ thuộc, theo bất kỳ cách nào, vào bất kỳ kiểu nào trong hai *kiểu con* mà nó sử dụng. Cả hai *kiểu con* đều có thể thay thế cho kiểu License.

Vấn đề Hình vuông/Hình chữ nhật

Pha kinh điển về vi phạm *Nguyên Tắc Thay Thế Liskov* là vấn đề *hình vuông/hình chữ nhật* nổi tiếng (hoặc khét tiếng, tùy theo quan điểm của bạn), được thể hiện trong hình dưới đây.



Trong ví dụ này, Square không nên là một kiểu con của Rectangle vì width và và height của Rectangle có thể thay đổi độc lập; ngược lại, width và height của Square phải thay đổi cùng nhau. Nếu *User* tin rằng đối tượng đang dùng là một Rectangle, họ có thể sẽ rối tinh lên. Đoạn mã sau cho thấy tại sao:

```
Rectangle r = …
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

Nếu r ở trên là một Square thì phép *assert* sẽ nắm chắc thất bại. Với các sắp đặt quan hệ kế thừa như hiện tại, cách duy nhất để khử sự vi phạm *Nguyên Tắc Thay Thế Liskov* là thêm các cơ chế để *User* có thể phát hiện xem r có phải là một Square hay không. Điều này sẽ khiến hành vi của *User* bị phụ thuộc vào kiểu dữ liệu mà nó sử dụng, và mất khả năng thay thế sang các kiểu dữ liệu khác.

Tác động mở rộng lên kiến trúc

Nguyên Tắc Thay Thế không chỉ như một hướng dẫn để dẫn dắt các mối quan hệ kế thừa. Nó còn có ảnh hưởng thấy rõ đến kiến trúc của hệ thống trên khía cạnh các *giao diện* và các *triển khai* của chúng.

Các *giao diện* có thể nằm ở nhiều dạng thức. Có thể là một giao diện Java, được triển khai bởi một số lớp; hay dưới dạng một lớp Ruby với khung các phương thức; hoặc cũng có thể là một tập hợp các dịch vụ đáp ứng cùng một giao diện REST… Dù là ở tình huống nào, *Nguyên Tắc Thay Thế Liskov* được áp dụng vì có những *người dùng* cần đến những giao diện được thiết kế tốt cũng như triển khai của chúng.

Cách tốt nhất để hiểu *Nguyên Tắc Thay Thế Liskov* từ quan điểm kiến trúc là xem xét những gì xảy ra với kiến trúc của một hệ thống khi nguyên tắc bị vi phạm.

Dịch vụ gọi xe

Giả sử chúng ta đang xây dựng một cổng tích hợp cho nhiều dịch vụ gọi xe công cộng. Khách hàng sử dụng cổng của chúng ta để tìm phương tiện đi lại phù hợp nhất, không kể là từ công ty nào. Khi khách đã có quyết định, chúng ta sẽ *chuyển hướng* yêu cầu của họ đến *tài xế* thông qua một *dịch vụ RESTful*.

Bây giờ giả sử rằng URI của dịch vụ chuyển tiếp được đặt trong cơ sở dữ liệu các *tài xế*. Một khi hệ thống của chúng ta chọn được trình *tài xế* phù hợp với yêu cầu của người dùng, nó sẽ lấy URI từ cơ sở dữ liệu của tài xế và sau đó thực hiện *chuyển tiếp*.

Giả sử tài xế *Bob* tại công ty *Purple Cab* có URI như sau:

```
purplecab.com/do/Bob
```

Hệ thống của chúng ta sẽ nối thông tin từ khách hàng vào URI này và PUT:

```
purplecab.com/do/Bob
    /pickupAddress/24 Maple St.
    /pickupTime/153
```

```
/destination/ORD
```

Dễ thấy rằng như vậy, tất cả các dịch vụ chuyển tiếp, bất kể công ty nào, đều sẽ có giao diện REST giống nhau. Chúng đều cần các tham số pickupAddress, pickupTime và destination.

Bây giờ, giả sử lập trình viên của công ty taxi *Acme* không đọc kỹ tài liệu. Họ viết tắt tham số destination. *Acme* là khách hàng lớn nhất của chúng ta và bà ngoại của CEO Acme là vợ mới của CEO của chúng ta, blah blah đại loại thế. Vậy là chúng ta phải thay đổi hệ thống của mình.

Rõ ràng là chúng ta cần thêm một trường hợp ngoại lệ. Yêu cầu chuyển tiếp đến trình điều khiển *Acme* sẽ phải sử dụng một bộ quy tắc khác với các trình điều khiển khác.

Và thế là một chỉ lệnh rẽ nhánh xuất hiện.

```
if (driver.getDispatchUri().startsWith("acme.com")) {
 // ...
```

Không một kiến trúc sư hệ thống nào xứng đáng với số muối mình từng ăn vào sẽ cho phép một chỉ lệnh như vậy tồn tại trong hệ thống. Sự tồn tại của từ acme đặt mã nguồn trước vô số lỗi nguy hiểm và bí ẩn, đấy là chưa kể tới các rủi ro an ninh.

Thử nghĩ điều gì sẽ xảy ra nếu *Acme* mua về *Purple Cab*, thống nhất tất cả các hệ thống, nhưng vẫn duy trì thương hiệu và website riêng biệt? Chẳng lẽ lại phải phải bổ sung thêm một chỉ lệnh rẽ nhánh khác?

Kiến trúc sư của chúng ta sẽ tìm phương án cách ly hệ thống khỏi các vấn đề như thế này, bằng cách tạo ra một loại mô-đun-kiến-tạo-yêu cầu-chuyển-tiếp, được điều khiển bởi một cơ sở dữ liệu có khả năng cấu hình trông như dưới đây:

```
URI             Dispatch Format
---
Acme.com        /pickupAddress/%s/pickupTime/%s/dest/%s
*.*             /pickupAddress/%s/pickupTime/%s/destination/%s
```

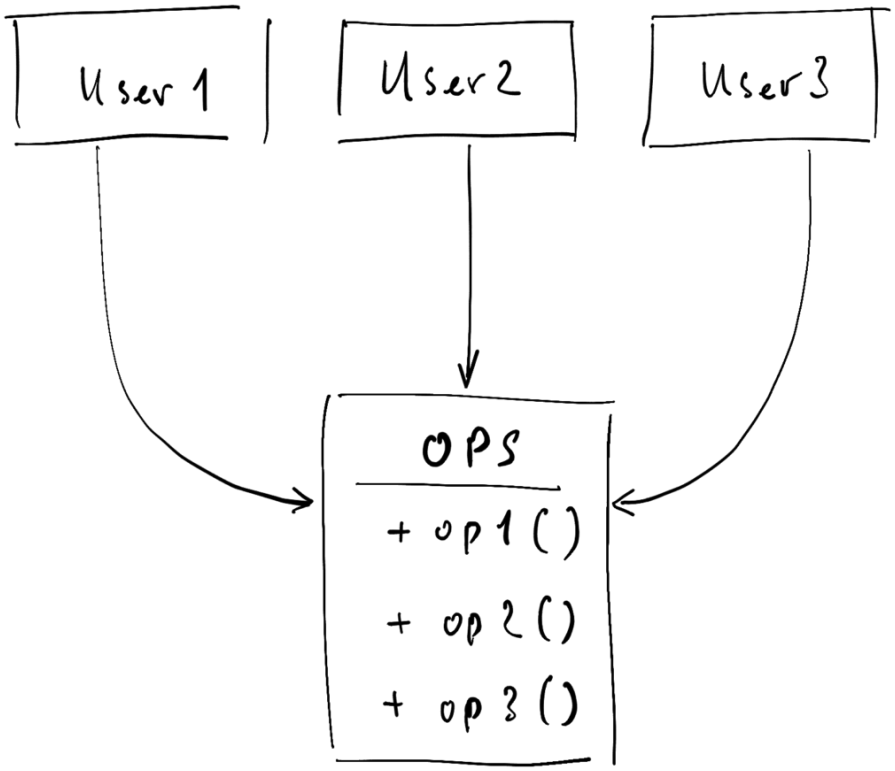
Và thế là anh ta phải đối phó thêm với một lượng lớn thấy rõ các cơ chế và mô-đun mới. Chỉ vì có một dịch vụ con không tuân thủ nguyên tắc thay thế.

Kết luận

Nguyên Tắc Thay Thế Liskov có thể, và nên được áp dụng tại mức kiến trúc. Một vi phạm đơn giản về khả năng thay thế có thể khiến kiến trúc của hệ thống bị xâm nhiễm bởi một lượng lớn các cơ chế bổ sung.

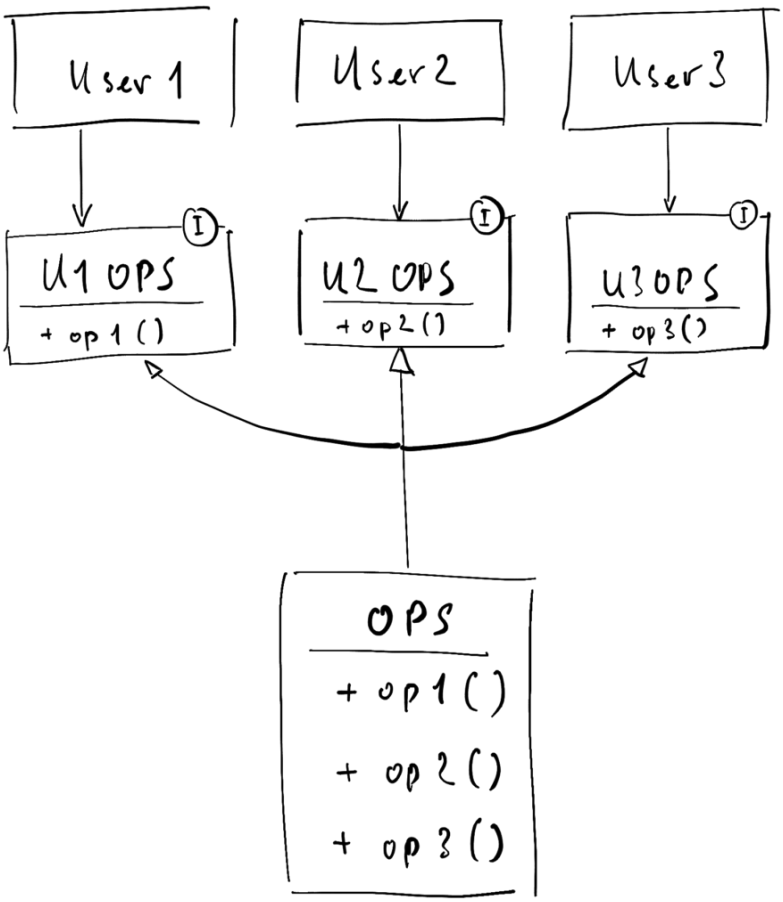
Nguyên Tắc Phân Tách Giao Diện

Nguyên Tắc Phân Tách Giao Diện được lấy tên từ sơ đồ dưới đây:



Trong tình huống trên, có một số người dùng sử dụng đến các hoạt động của OPS. Giả sử rằng *User1* sử dụng đến `op1`, *User2* sử dụng `op2` và *User3* sử dụng `op3`.

Bạn cứ thử nghĩ mà xem, thế rồi mã nguồn của *User1* sẽ phụ thuộc vào `op2` và `op3`, mặc dù nó không gọi chúng. Vấn đề này có thể giải quyết bằng cách tách các hoạt động của OPS vào các giao diện riêng như dưới đây:



Một trường hợp khác, tương tự, nhưng ở cấp độ cao hơn, khi hệ thống của bạn phụ thuộc vào một framework *F*, và *F* thì bị ràng buộc với một cơ sở dữ liệu *D*. Điều này đã khiến cho hệ thống của bạn bị ràng buộc với *D*, mặc dù có khi bạn chẳng hề dùng đến.

Những phụ thuộc như thế có thể gây ra điều gì xấu? Mã nguồn bị buộc phải biên dịch lại khi các mối phụ thuộc có thay đổi là điều thứ nhất. Thứ hai, một khi các phụ thuộc có vấn đề, vấn đề đó có thể gây ra lỗi ở mô-đun của bạn, mặc dù bạn không dùng đến bất kỳ tính năng nào ở đó.

Kết luận

Bài học rút ra ở đây là những mối phụ thuộc không cần thiết có thể gây ra những rắc rối không ngờ tới. Hãy tránh những mối phụ thuộc đó.

Nguyên Tắc Đảo Ngược Phụ Thuộc

Nguyên Tắc Đảo Ngược Phụ Thuộc cho chúng ta biết rằng các hệ thống linh hoạt nhất là những hệ thống trong đó các phụ thuộc mã nguồn chỉ đề cập đến *trừu tượng*, chứ không phải các *cụ thể hóa*.

Trong những ngôn-ngữ-định-kiểu-tĩnh, Java chẳng hạn, phát biểu trên có nghĩa là các chỉ lệnh use, import, hay include chỉ nên tham chiếu đến các mô-đun chứa giao diện, lớp trừu tượng hay các định nghĩa trừu tượng khác. Không nên tham chiếu đến bất kỳ thứ gì *cụ thể*.

Quy tắc tương tự cũng được áp dụng cho các ngôn-ngữ-định-kiểu-động, như Ruby hay Python. Các phụ thuộc mã nguồn không nên tham chiếu đến các mô-đun cụ thể. Tuy nhiên trong các ngôn ngữ này, việc phân định rõ một mô-đun có được gọi là *cụ thể* hay không khó khăn hơn đôi chút. Mô-đun cụ thể là những mô-đun mà các hàm của nó đã được triển khai.

Rõ ràng, coi phát biểu trên là một quy tắc là phi thực tế. Các hệ thống phần mềm phải phụ thuộc vào nhiều công cụ cụ thể. Ví dụ, lớp *String* trong Java là một lớp cụ thể, và sẽ không tố chút nào nếu cố ép *String* trở thành lớp trừu tượng. Không thể tránh phụ thuộc mã nguồn vào lớp java.lang.string, và không nên tránh.

Khi cân nhắc cẩn thận, lớp *String* rất ổn định. Nó rất hiếm khi thay đổi và các thay đổi đó luôn được kiểm soát chặt chẽ. Lập trình viên và các kiến trúc sư không phải lo lắng về những thay đổi thường xuyên và thất thường của *String*.

Với những lý do tương tự, chúng ta nên bỏ qua những hoạt động ổn định của hệ điều hành cũng như các công cụ nền tảng khi nói đến Nguyên Tắc Đảo Ngược Phụ Thuộc. Chúng ta chấp nhận những phụ thuộc đó bởi chúng ta có thể tin tưởng chúng sẽ không thay đổi.

Thứ cần lưu tâm là những triển khai cụ thể trong hệ thống của chúng ta. Đó là những mô-đun mà chúng ta đang tích cực phát triển và đang trải qua thay đổi thường xuyên.

Những thành phần trừu tượng ổn định

Mỗi thay đổi trên một giao diện *trừu tượng* luôn kéo theo thay đổi trên những triển khai *cụ thể* của nó. Ngược lại, các thay đổi của triển khai *cụ thể* không phải lúc nào cũng (thậm chí thường là không) yêu cầu thay đổi giao diện mà chúng triển khai. Do đó, các giao diện ít biến đổi hơn so với các triển khai.

Các nhà thiết kế phần mềm và kiến trúc sư giỏi sẽ cố gắng để giảm thiểu sự biến đổi của các giao diện. Họ sẽ cố gắng tìm cách thêm chức năng vào triển khai mà không cần phải thay đổi giao diện. Điều này nằm trong mọi bài học vỡ lòng về thiết kế.

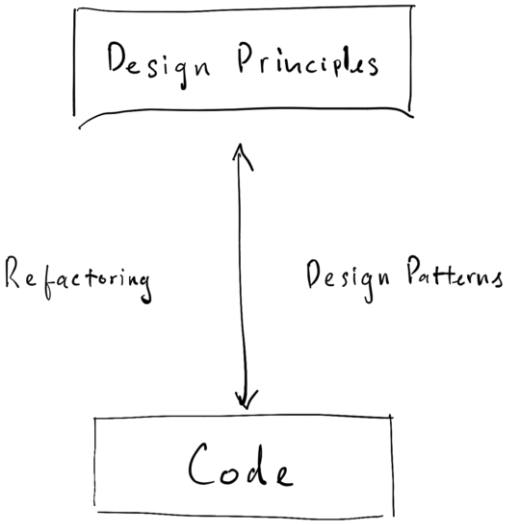
Như vậy là muốn kiến trúc phần mềm được ổn định thì cần tránh đi sự phụ thuộc vào những gì *cụ thể* và dễ thay đổi. Điều này kéo theo một danh sách các hành dụng khi viết mã:

- **Không tham chiếu các lớp *cụ thể* dễ thay đổi**. Thay vào đó, tham chiếu đến các giao diện trừu tượng. Quy tắc này áp dụng cho tất các các ngôn ngữ, cho dù được định kiểu tĩnh hay động. Nó cũng đặt ra những hạn chế khắt khe cho thao tác tạo ra các đối tượng và thường bắt buộc sử dụng mẫu thiết kế *Abstract Factory*.
- **Không kế thừa các lớp *cụ thể* dễ thay đổi**. Đây thật ra là một hệ quả của quy tắc trên, nhưng được viết ra tường minh để nhấn mạnh. Trong các ngôn ngữ định kiểu tĩnh, kế thừa là mạnh mẽ và cứng nhắc nhất trong số tất cả các mối quan hệ trong mã nguồn; do đó nó nên được sử dụng thật cẩn thận. Trong các ngôn ngữ định kiểu động, sự kế thừa ít gặp vấn đề hơn, nhưng **nó vẫn là một sự phụ thuộc**, và sử dụng phụ thuộc một cách cẩn trọng luôn là một hành động khôn ngoan.
- **Không ghi đè các hàm *cụ thể***. Các hàm cụ thể thường mang theo các phụ thuộc mã nguồn. Khi ghi đè một hàm, chúng ta cũng kế thừa luôn các mối phụ thuộc của nó. Nếu muốn quản lý được các phụ thuộc của hàm, chúng ta nên khiến hàm trở thành hàm trừu tượng và sau đó kiến tạo nhiều triển khai khác nhau.
- Không bao nhắc đến tên của bất kỳ thứ gì *cụ thể* và dễ thay đổi. Đây thực ra là hồi quy của chính nguyên tắc đầu tiên.

Kết luận

Chúng ta sẽ dành 10% thời gian đầu tiên của dự án để viết nên 90% tính năng. Sau đó chúng ta sẽ dành 90% thời gian còn lại để viết 10% tính năng cuối cùng (mà không biết có xong không).

Các nguyên tắc SOLID không phải là những hướng dẫn thực hành, chúng cũng không được viết ra bởi tính quân phiệt hay kể cả đồng thuận. SOLID xuất phát từ các vấn đề thực tế, rằng sự vi phạm chúng sẽ ngay lập tức dẫn đến những hệ quả rất xấu cho phần mềm. Sự vi phạm các nguyên tắc thiết kế sẽ khiến cho **phần mềm ngày càng trở nên khó bổ sung tính năng mới hơn**.



Khoảng cách giữa mã nguồn và các nguyên tắc thiết kế được lấp đầy thông qua *tái cấu trúc*, hoặc áp dụng các *mẫu thiết kế*, hoặc cả hai. Tái cấu trúc giúp phát hiện mà gỡ bỏ những mã có "mùi" – mà thường là dấu hiệu của một bộ phận kiến trúc không tốt. Trong khi đó các mẫu thiết kế đưa ra những giải pháp hoàn thiện và ổn định để tạo nên các thiết kế cục bộ mà thỏa mãn các nguyên tắc thiết kế.

Bài 7. Sự khác biệt giữa thiết kế tốt và thiết kế tồi trong kỹ thuật phần mềm

Thuật ngữ thiết kế được sử dụng theo **hai cách**. Trong khi được sử dụng như động từ, nó có nghĩa là - quá trình thiết kế và được sử dụng như danh từ, nó có nghĩa là - kết quả của quá trình thiết kế. Thiết kế là một mô tả kỹ thuật ít ỏi của một thứ gì đó sẽ được xây dựng. Kết quả của một quá trình thiết kế được gọi là mô hình thiết kế hoặc thiết kế của hệ thống.

Thiết kế cho phần mềm máy tính cũng quan trọng như một lộ trình cho một ngôi nhà, nếu không kết quả sẽ là sự hỗn loạn. Thiết kế hệ thống là một kế hoạch cho một giải pháp cho hệ thống. Ở đây, một hệ thống là các thành phần có hành vi xác định rõ ràng tương tác với nhau theo cách được xác định trước để tạo ra các hành vi hoặc dịch vụ nhất định cho môi trường của nó.

Quá trình thiết kế phần mềm thường có hai cấp độ. Một trong số đó xác định các mô-đun cho hệ thống, các thông số kỹ thuật của chúng và mối quan hệ qua lại giữa chúng. Đây được gọi là thiết kế phức tạp hoặc thiết kế logic, quyết định thiết kế bên trong của mô-đun.

Phương pháp luận thiết kế là một phương pháp có hệ thống tạo ra một thiết kế bằng cách áp dụng một tập hợp các kỹ thuật và hướng dẫn. Hầu hết các thực hành thiết kế tập trung vào thiết kế hệ thống. Cũng cung cấp một bộ hướng dẫn để giúp nhà phát triển, thiết kế hệ thống.

Bảng phân biệt thiết kế tốt và tồi sau đây dựa trên các đặc điểm chính của nó.

Tiêu chí	Thiết kế tốt	Thiết kế tồi
Thay đổi	Thay đổi là một phần của hệ thống không phải lúc nào cũng đòi hỏi sự thay đổi ở phần khác của hệ thống.	Một thay đổi đòi hỏi phải thay đổi nhiều phần của hệ thống.
Logic	Mỗi phần logic chỉ viết một lần.	Logic có thể được nhân bản ở nhiều chỗ.
Tính tự nhiên	Đơn giản	Phức tạp
Chi phí	Nhỏ	Lớn
Các liên kết	Liên kết logic có thể dễ dàng tìm thấy	Liên kết logic không thể nhớ
Tính mở rộng	Hệ thống có thể được mở rộng với những thay đổi chỉ ở một nơi.	Hệ thống không thể mở rộng một cách dễ dàng.

Bài 8. Giải thích mô hình MVC thông qua ... cốc trà đá

Nếu bạn từng đi uống trà đá, thì bạn đã hiểu được MVC rồi

Model - View - Controller (MVC) là một mô hình thiết kế web hiện đại đã trở nên quá đỗi quen thuộc. Cứ thử bước vào một phòng làm việc của các lập trình viên xem, bạn sẽ bị "dội bom" bởi hàng đống thứ khủng khiếp mà bạn có khi còn chưa từng nghe tên như là Ruby on Rails, Angular hay Django.

Nhìn rộng hơn, logic của mô hình MVC được sử dụng để mô tả quá trình làm web của đại đa số các ngôn ngữ như là PHP, Ruby, Python hay JavaScript.

Nhiều ông lập trình web mà cứ theo kiểu nhìn đời qua kẽ lá. Cứ thử để mấy ông lập trình viên khác nhìn vào code của họ xem, họ sẽ cho ngay một bài thuyết trình dài cả tiếng toàn những giáo điều quen thuộc.

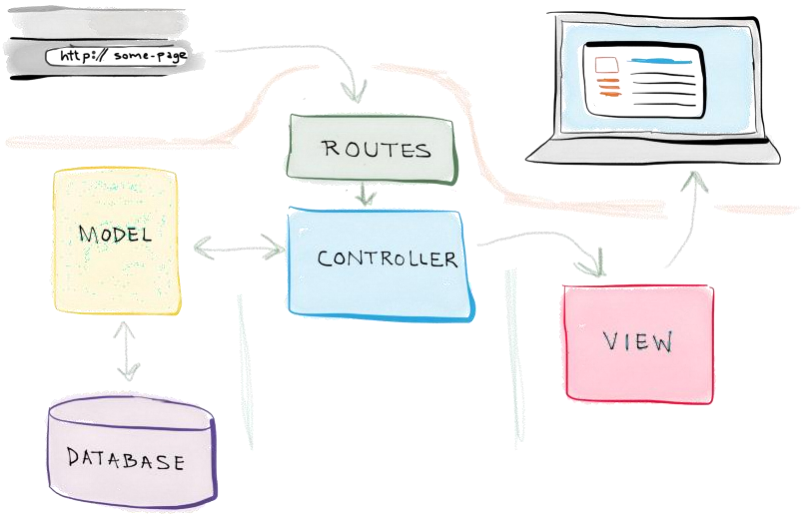
Mà rõ ràng là mấy thứ đấy không thực tế. Trong khi mô hình MVC hiện tại hoàn toàn có thể được giải thích dễ dàng bằng hình thức trà đá vỉa hè. Vậy nên, nếu bạn từng đi ngồi trà đá với bạn bè, thì bạn có thể hiểu được cấu trúc của các ứng dụng web hiện đại rồi đấy.

Mô hình MVC là gì?

MVC gồm 3 thành phần bao gồm:

- M là Model: một cấu trúc dữ liệu chắc chắn, chuẩn bị dữ liệu để cung cấp cho Controller
- V là View: nơi hiển thị dữ liệu cho người dùng theo cách mà người dùng có thể dễ dàng hiểu và tương tác được
- C là Controller: nhận về lệnh từ người dùng, gửi lệnh đến Model để lấy hay cập nhật dữ liệu, rồi truyền lệnh đến View để cập nhật giao diện hiển thị cho đúng với dữ liệu đã cập nhật

Trên sơ đồ thì nó sẽ như thế này:



Ôi giời lằng nhằng phức tạp quá. Làm cốc trà đá cho hạ nhiệt

Bạn bước vào quán trà đá. Giữa mùa hè, quán nào quán nấy nóng nực đông đúc. Bạn luồn lách qua đám đông để gọi cho bằng được bà chủ quán "cho cháu một cốc trà đá cô ơi".

Lúc này, bạn là "người dùng" và "cốc trà đá" là "yêu cầu từ phía người dùng". Đối với bạn, cốc trà đá là thứ đồ uống ưa thích, mát lạnh, quá phù hợp để xua tan cái nóng thủ đô 40 độ C.

Bà chủ quán gật đầu "ô kê em nhớ". Đối với bà chủ quán, cốc trà đá không chỉ ngon lành mát lạnh, mà còn là một mớ quy trình các bước:

1. Lấy cái cốc
2. Cho đá vào
3. Cho trà vào
4. Cho thêm nước lọc cho trà loãng bớt
5. Khuấy đều lên
6. Đưa cốc trà cho bạn
7. Thanh toán

Bộ não của bà chủ quán lúc này đóng vai trò Controller. Kể từ thời điểm bạn nói "một cốc trà đá" bằng tiếng Việt và bà chủ quán hiểu được, công việc bắt đầu. Trà đá, nước mía hay cocktail thì cũng như nhau, nhưng nguyên liệu thì hoàn toàn khác biệt. Bà chủ quán chỉ có thể sử dụng những công cụ và nguyên liệu của quán, và những công cụ đó sẽ đóng vai trò Model, bao gồm:

- Đôi tay của bà chủ
- Các nguyên liệu pha chế (trà, nước ...)
- Đá lạnh

- Bia, nước ngọt, thuốc lá...
- Chanh, sấu, gừng...
- Các ly cốc để đựng đồ uống

Những nguyên liệu và công cụ này, thông qua một loạt các bước, đã trở thành cốc trà đá mát lạnh đến tay bạn. Cốc trà đá lúc này đóng vai trò "View". "View" được làm nên từ những công cụ, nguyên liệu trong "Model", được chế biến và bàn giao tới tay bạn thông qua "Controller" (chính là bộ não của bà chủ quán)

Bài học rút ra là gì?

- 1 cốc không đủ, bạn muốn gọi cốc nữa? Rõ ràng là bạn không thể hét to vào cái cốc đã hết (chính là "View") được, bạn phải gọi bà chủ quán "Controller".
- Thời gian từ lúc bà chủ quán nhận được yêu cầu tới khi làm xong phải tối thiểu nhất có thể. Đó chính là "skinny controller", có thể hiểu là "controller" nên chứa tối thiểu lượng logic cần xử lý và được quản lý lượng model nhiều nhất có thể. Một bà chủ quán "thiện nghệ" không chỉ ghi nhớ chính xác cách làm, mà còn chuẩn bị đầy đủ công cụ và nguyên liệu để không mất nhiều thời gian tìm kiếm và chế biến.
- Vậy nếu bà chủ quán đưa hết nguyên liệu cho bạn rồi bảo bạn tự pha? Chẳng ai làm thế cả. Do vậy, bạn cần phải để việc xử lý logic trên model nhiều nhất có thể, và tối giản hóa view. Nói cách khác, được phục vụ tận miệng thì vẫn thích hơn là phải đi pha.
- Nếu bạn gọi 1 lon bia thì sao? Bà chủ chắc chẳng phải làm gì nhiều, bật nắp lon bia rồi đưa bạn là xong. Nhưng mà rõ ràng bạn vẫn phải gọi bà chủ quán, vì lon bia không thể tự nhảy ra trước mặt bạn được.

Quay trở lại vấn đề lập trình web

Giờ thử xem một ứng dụng web hiện đại có quy trình thế nào nhé

- Người dùng tạo ra một yêu cầu thông qua đường dẫn, ví dụ /home
- Controller nhận yêu cầu và đưa ra một mệnh lệnh để xử lý yêu cầu đó. Nếu lệnh thực thi với phần View thì cập nhật lại màn hình hiển thị, với Model thì để trình diễn logic. Giả sử yêu cầu của người dùng có yếu tố logic
- Model thực thi phần logic lấy từ một cơ sở dữ liệu nào đó và gửi trả lại phản hồi theo hướng dẫn từ Controller
- Controller truyền dữ liệu này ra phần view và cập nhật lại giao diện cho người dùng.
- Mọi yêu cầu đều phải đi qua Controller trước khi chuyển hóa thành lệnh thực thi cho View hay Model.

Tổng kết

Bất cứ khi nào bạn học một framework lập trình web mới, bạn sẽ gặp mô hình MVC. Nói cách khác, một khi bạn đã lập trình dựa trên MVC thì không cần phải ngán bất cứ framework mới nào cả.

Bài 9. Sự khác biệt giữa các mẫu thiết kế MVC và MVT

Mẫu thiết kế MVC (Model-View-Controller)

Đây là một mẫu thiết kế phần mềm được sử dụng để triển khai các giao diện người dùng và nhấn mạnh vào việc tách phần biểu diễn dữ liệu khỏi các thành phần tương tác và xử lý dữ liệu.

Mẫu thiết kế MVC có 3 thành phần và mỗi thành phần có một mục đích cụ thể:

- Model: Đây là thành phần trung tâm của kiến trúc và quản lý dữ liệu, logic cũng như các ràng buộc khác của ứng dụng.
- View: Đây là thành phần đề cập đến cách dữ liệu sẽ được hiển thị cho người dùng và cung cấp các thành phần biểu diễn dữ liệu khác nhau.
- Controller: Đây là thành phần điều khiển Model và hiển thị View bằng cách đóng vai trò là cầu nối giữa cả hai.

Mẫu thiết kế MVT (Model-View-Template):

Đây là một mẫu thiết kế khác tương tự như MVC. Nó cũng được sử dụng để triển khai các ứng dụng và giao diện web nhưng ngược lại với MVC, phần controller được chính framework đảm nhận cho chúng ta.

Nó có 3 thành phần và mỗi thành phần có một mục đích cụ thể:

Model: Tương tự như MVC, thành phần này hoạt động như một giao diện cho dữ liệu của bạn và về cơ bản là cấu trúc logic đằng sau toàn bộ ứng dụng web được đại diện bởi cơ sở dữ liệu như MySql, PostgreSQL.

- **View:** Thực thi logic nghiệp vụ và tương tác với Model và render template. Nó chấp nhận HTTP request và sau đó trả lại các HTTP response.
- **Template:** Đây là thành phần làm cho MVT khác với MVC. Các template đóng vai trò như lớp trình bày (presentation) và về cơ bản là mã HTML hiển thị dữ liệu. Nội dung trong các tệp này có thể là tĩnh hoặc động.

Bảng sau sẽ so sánh sự khác nhau giữa mẫu thiết kế MVC và MVT:

STT	MODEL VIEW CONTROLLER (MVC)	MODEL VIEW TEMPLATE (MVT)
1.	MVC có Controller điều khiển cả Model và View.	MVT có View để nhận HTTP request và trả lại HTTP response.
2.	View cho biết dữ liệu người dùng sẽ được trình bày như thế nào	Template được sử dụng trong MVT cho mục đích tương tự
3.	Trong MVC, chúng ta phải viết tất cả mã Controller cụ thể.	Phần Controller được quản lý bởi chính Framework.
4.	Kết hợp chặt chẽ	Kết hợp lỏng lẻo
5.	Khó sửa đổi	Dễ sửa đổi
6.	Thích hợp phát triển cho những ứng dụng lớn	Thích hợp cho phát triển ứng dụng lớn và nhỏ

7.	Luồng được xác định rõ ràng do đó dễ hiểu.	Luồng đôi khi khó hiểu hơn so với MVC.
8.	Nó không liên quan đến ánh xạ các URL.	Ánh xạ mẫu URL diễn ra.
9.	Các ví dụ như: ASP.NET MVC, Spring MVC v..v..	Django sử dụng mẫu MVT.

Bài 10. Kiến trúc lục giác trong Java

Theo nguyên tắc thiết kế phát triển phần mềm, phần mềm yêu cầu nỗ lực bảo trì tối thiểu được coi là thiết kế tốt. Có nghĩa là, bảo trì phải là điểm mấu chốt mà một kiến trúc sư phần mềm cần phải xem xét. Trong bài viết này, một kiến trúc như vậy, được gọi là Kiến trúc lục giác giúp phần mềm dễ bảo trì, quản lý, kiểm tra và mở rộng quy mô sẽ được thảo luận.

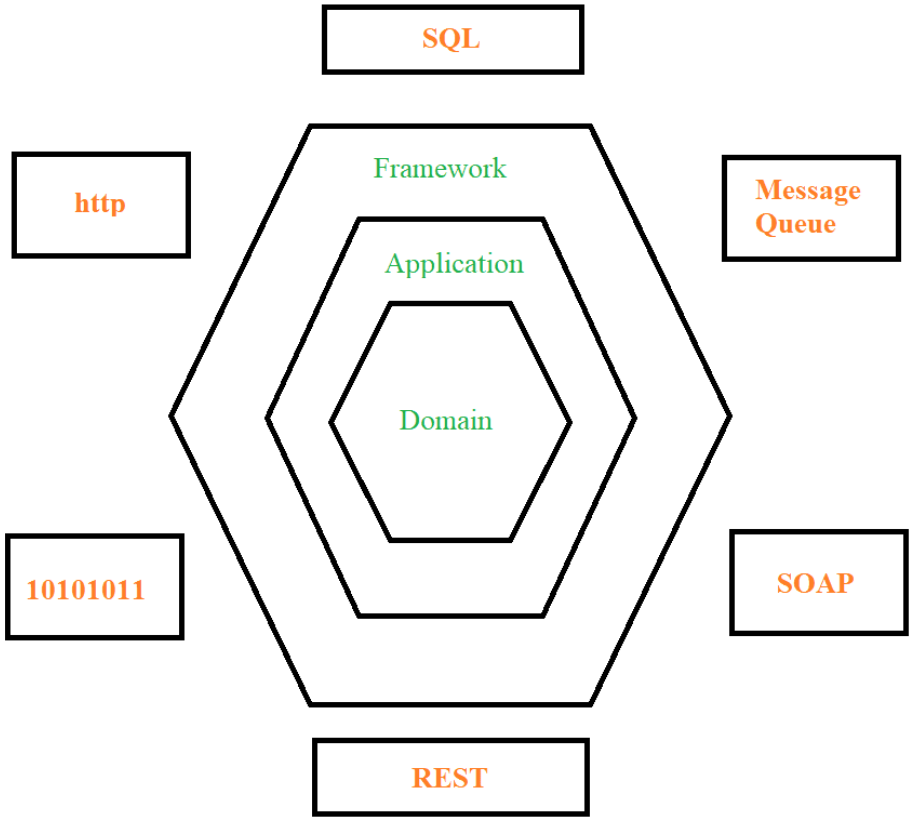
Kiến trúc lục giác là một thuật ngữ do Alistair Cockburn đặt ra vào năm 2006. Tên gọi khác của kiến trúc Lục giác là kiến trúc Ports And Adapters. Kiến trúc này chia một ứng dụng thành hai phần cụ thể là phần bên trong và phần bên ngoài. Logic cốt lõi của một ứng dụng được coi là phần bên trong. Database, UI và Message Queue có thể là phần bên ngoài. Khi làm như vậy, logic ứng dụng cốt lõi đã được cách ly hoàn toàn với thế giới bên ngoài. Bây giờ giao tiếp giữa hai phần này có thể xảy ra thông qua Cổng (Port) và Bộ điều hợp (Adapters). Bây giờ, chúng ta hãy tìm hiểu ý nghĩa của mỗi điều này.

Các cổng (Ports): Hoạt động như một cổng mà qua đó giao tiếp diễn ra như một cổng vào hoặc ra. Một cổng Inbound là một thứ giống như một giao diện dịch vụ (Service Interface) thể hiện logic cốt lõi với thế giới bên ngoài. Một cổng Outbound là một cái gì đó giống như một giao diện kho lưu trữ (Repository Interface) tạo điều kiện giao tiếp từ ứng dụng đến hệ thống lưu trữ (Persistence System).

Bộ điều hợp (Adapters): Bộ điều hợp hoạt động như một triển khai của một cổng xử lý đầu vào của người dùng và dịch nó thành lời gọi theo ngôn ngữ cụ thể. Về cơ bản, nó đóng gói logic để tương tác với các hệ thống bên ngoài như Message Queue, database, v.v. Nó cũng chuyển đổi giao tiếp giữa các đối tượng bên ngoài và lõi. Bộ điều hợp lại có hai loại.

- **Bộ điều hợp chính (Primary Adapters):** Nó điều khiển ứng dụng bằng cách sử dụng cổng đến của ứng dụng và còn được gọi là Driving Adapters. Ví dụ về bộ điều hợp chính có thể là WebViews hoặc Rest Controllers.
- **Bộ điều hợp thứ cấp (Secondary Adapters):** Đây là một triển khai của một cổng ra ngoài được ứng dụng điều khiển và còn được gọi là bộ điều hợp điều khiển. Kết nối với Message Queue, database và lệnh gọi API bên ngoài là một số ví dụ về Bộ điều hợp thứ cấp.

Do đó, kiến trúc lục giác nói về việc hiển thị nhiều điểm cuối trong một ứng dụng cho mục đích giao tiếp. Nếu chúng ta có bộ điều hợp phù hợp cho cổng của mình, yêu cầu của chúng ta sẽ được giải quyết. Kiến trúc này là một kiến trúc phân lớp và chủ yếu bao gồm ba lớp, Khung (Framework), Ứng dụng (Application) và Miền (Domain).



Miền (Domain): Nó là một lớp logic nghiệp vụ cốt lõi và các chi tiết triển khai của các lớp bên ngoài được ẩn với lớp này.

Ứng dụng (Application): Nó hoạt động như một trung gian giữa lớp Miền và lớp Khung.

Khung (Framework): Lớp này có tất cả các chi tiết triển khai mà một lớp miền sẽ tương tác với thế giới bên ngoài như thế nào.

Ví dụ minh họa: Hãy hiểu kiến trúc này bằng một ví dụ thời gian thực. Chúng ta sẽ thiết kế một ứng dụng Cake Service bằng Spring Boot. Bạn cũng có thể tạo một dự án dựa trên Spring hoặc Maven bình thường, tùy thuộc vào sự thuận tiện của bạn. Sau đây là các phần khác nhau trong ví dụ:

Miền: Cốt lõi của ứng dụng. Tạo một lớp Cake với các thuộc tính của nó, để đơn giản, chúng ta sẽ thêm tên vào đây.

```
// Consider this as a value object
// around which the domain logic revolves.
public class Cake implements Serializable {

	private static final long serialVersionUID
		= 100000000L;
	private String name;

	// Getters and setters for the name
	public String getName()
	{
		return name;
	}

	public void setName(String name)
	{
		this.name = name;
	}

	@Override
	public String toString()
	{
		return "Cake [name=" + name + "]";
	}
}
```

Cổng đến (Inbound Port): Xác định giao diện mà qua đó ứng dụng cốt lõi của chúng ta sẽ kích hoạt giao tiếp của nó. Nó đưa ứng dụng cốt lõi ra thế giới bên ngoài.

```
import java.util.List;
// Interface through which the core
// application communicates. For
// all the classes implementing the
// interface, we need to implement
// the methods in this interface
public interface CakeService {

    public void createCake(Cake cake);

    public Cake getCake(String cakeName);
```

```
    public List<Cake> listCake();
}
```

Cổng đi (Outbound Port): Tạo thêm một giao diện để tạo hoặc truy cập thế giới bên ngoài.

```
import java.util.List;

// Interface to access the cake
public interface CakeRepository {

    public void createCake(Cake cake);

    public Cake getCake(String cakeName);

    public List<Cake> getAllCake();
}
```

Bộ điều hợp chính (Primary Adapter): Bộ điều khiển có thể là bộ điều hợp chính của chúng ta, nó sẽ cung cấp các điểm cuối để tạo và tìm nạp tài nguyên.

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

// This is the REST endpoint
@RestController
@RequestMapping("/cake")
public class CakeRestController implements CakeRestUI {
@Autowired
    private CakeService cakeService;

    @Override
    public void createCake(Cake cake)
    {
        cakeService.createCake(cake);
    }

    @Override
    public Cake getCake(String cakeName)
    {
        return cakeService.getCake(cakeName);
    }
```

Chúng ta có thể tạo thêm một giao diện cho **CakeRestUI** như sau:

```
import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
```

```
public interface CakeRestUI {
    @PostMapping
    void createCake(@RequestBody Cake cake);

    @GetMapping("/{name}")
    public Cake getCake(@PathVariable String name);

    @GetMapping
    public List<Cake> listCake();
}
```

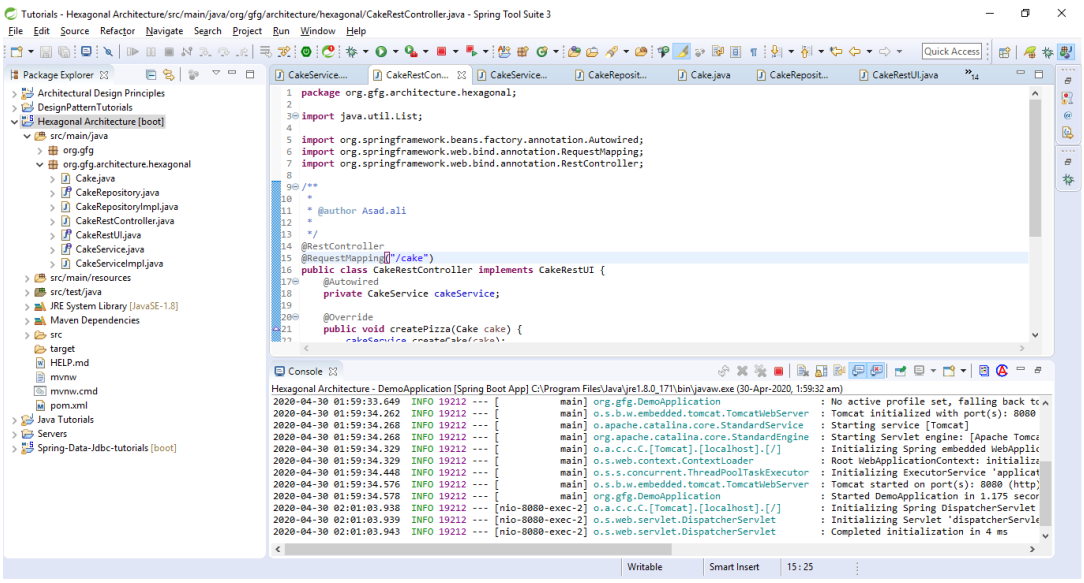
Bộ điều hợp thứ cấp (Secondary Adapter): Đây sẽ là việc triển khai một cổng đi. Vì CakeRepository là cổng gửi đi của chúng ta, vì vậy hãy triển khai nó.

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

import org.springframework.stereotype.Repository;

// Implementing the interface and
// all the methods which have been
// defined in the interace
@Repository
public class CakeRepositoryImpl
    implements CakeRepository {
    private Map<String, Cake> cakeStore
        = new HashMap<String, Cake>();

    @Override
    public void createCake(Cake cake)
    {
        cakeStore.put(cake.getName(), cake);
    }

    @Override
    public Cake getCake(String cakeName)
    {
        return cakeStore.get(cakeName);
    }
      @Override
    public List<Cake> getAllCake()
    {
        return
cakeStore.values().stream().collect(Collectors.toList());
    }
}
```

Giao tiếp giữa lõi với Nguồn dữ liệu: Cuối cùng, hãy tạo một lớp triển khai sẽ chịu trách nhiệm giao tiếp giữa ứng dụng lõi với nguồn dữ liệu bằng cách sử dụng một cổng ra.

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

// This is the implementation class
// for the CakeService
@Service
public class CakeServiceImpl
    implements CakeService {

    // Overriding the methods defined
    // in the interface
    @Autowired
private CakeRepository cakeRepository;

    @Override
    public void createCake(Cake cake)
    {
        cakeRepository.createCake(cake);
    }
    @Override
    public Cake getCake(String cakeName)
    {
        return cakeRepository.getCake(cakeName);
    }

    @Override
    public List<Cake> listCake()
    {
        return cakeRepository.getAllCake();
    }
}
```

Cuối cùng chúng ta đã triển khai tất cả các phương thức được yêu cầu trong ví dụ đã cho. Sau đây là kết quả khi chạy đoạn mã trên:



Bây giờ, hãy tạo một số Cake cho ví dụ trên bằng API REST. API sau được sử dụng để đẩy các Cake vào kho lưu trữ (repository). Vì chúng ta đang tạo và thêm dữ liệu, chúng ta sử dụng POST request. Ví dụ:

- **API:** [*POST*]: http://localhost:8080/cake

  Input Body

```
{
    "name" : "Black Forest"
}
```

- **API:** [*POST*]: http://localhost:8080/cake

  Input Body

```
{
    "name" : "Red Velvet"
}
```

- **API:** [*GET*]: http://localhost:8080/cake

  Output

```
[
    {
        "name": "Black Forest"
    },
    {
        "name": "Red Velvet"
    }
]
```

Ưu điểm của kiến trúc Lục giác:

- **Dễ bảo trì:** Vì logic ứng dụng cốt lõi (các lớp và đối tượng) được cách ly với thế giới bên ngoài và nó được liên kết lỏng lẻo, nên việc bảo trì sẽ dễ dàng hơn. Sẽ dễ dàng hơn để thêm một số tính năng mới vào một trong hai lớp mà không cần chạm vào lớp kia.
- **Dễ dàng điều chỉnh các thay đổi mới:** Vì tất cả các lớp đều độc lập và nếu chúng ta muốn thêm hoặc thay thế một cơ sở dữ liệu mới, chúng ta chỉ cần thay thế hoặc thêm các bộ điều hợp cơ sở dữ liệu mà không cần thay đổi miền logic của một ứng dụng.
- **Dễ dàng kiểm thử:** Việc kiểm thử trở nên dễ dàng. Chúng ta có thể viết các trường hợp thử nghiệm cho từng lớp bằng cách chỉ mô phỏng các cổng bằng bộ điều hợp giả (mock adapters).

Bài 11. Những Design Pattern thường dùng trong Android

Lời mở đầu

Trong quá trình thực hiện các dự án, ngoài việc làm thoả mãn yêu cầu của khách hàng, việc viết code một cách rõ ràng, sạch sẽ (clean code) là một điều vô cùng quan trọng.

Có thể trong tương lai bạn sẽ phải phát triển một chức năng mà kế thừa lại những code cũ củabạn, hoặc chí ít là trong quá trình maintain phải đọc lại code thì việc viết code một cách "sạch sẽ" giúp quá những công việc sau này trở nên dễ dàng và ít phát sinh bug hơn.

Design patterns là các giải pháp đã được tối ưu hóa, được tái sử dụng cho các vấn đề lập trình mà chúng ta gặp phải hàng ngày. Nó là một khuôn mẫu đã được suy nghĩ, giải quyết trong tình huống cụ thể rồi.

Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.

Nó không phải là ngôn ngữ cụ thể nào cả. Design Patterns có thể thực hiện được ở phần lớn các ngôn ngữ lập trình. Ta thường gặp nó nhất trong lập trình OOP.

Trong bài viết này tôi xin giới thiệu một số design pattern thường sử dụng trong Android.

Phân loại design pattern

Có 3 nhóm chính sau:

- *Creational Pattern* (nhóm khởi tạo) như *Builder, Dependency Injection, Singleton*… Nó sẽ giúp bạn trong việc khởi tạo đối tượng.
- *Structural Pattern* (nhóm cấu trúc) như *Adapter, Facade*… Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.
- *Behavioral Pattern* (nhóm hành vi) như *Command, Observer, Model View Controller, Model View View Model*. Nhóm này dùng trong thực hiện các hành vi của đối tượng.

Ta sẽ đi tìm hiểu cụ thể từng loại design pattern.

Creational Pattern

Builder

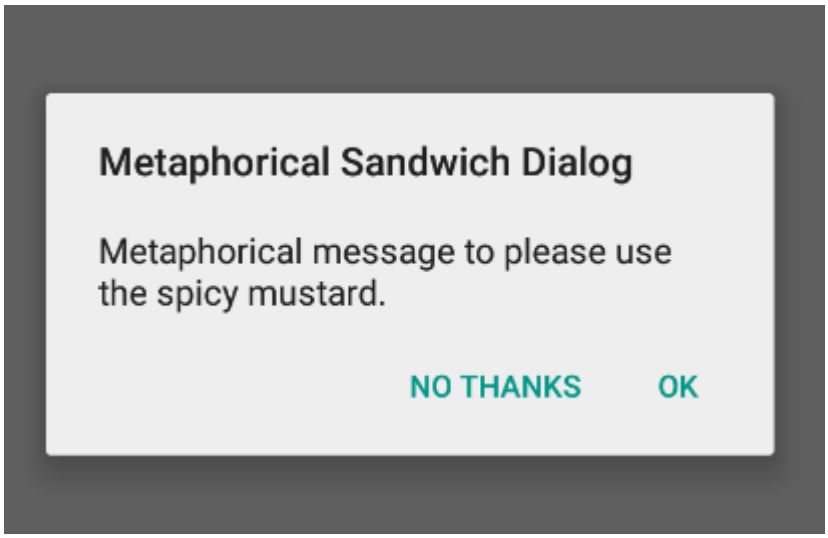
Builder design pattern phân chia việc khởi tạo (construction) tổng thể một đối tượng phức tạp ra làm nhiều phần nhỏ. Tức làthay vì tạo construction cho cả một đối tượng phức tạp thì nó sẽ tạoconstruction cho từng phần đặc trưng của đối tượng đó.

Trong Android, Builder design pattern xuất hiện khi sử dụngnhững đối tượng như AlertDialog.Builder.

```
new AlertDialog.Builder(this)
    .setTitle("Metaphorical Sandwich Dialog")
    .setMessage("Metaphorical message to please usethe spicy
mustard.")
    .setNegativeButton("No thanks",
newDialogInterface.OnClickListener() {
      @Override public void onClick(DialogInterface dialogInterface,
int i) {
        // "No thanks" button wasclicked
      }
    })
    .setPositiveButton("OK", new DialogInterface.OnClickListener() {
      @Override public void onClick(DialogInterface dialogInterface,
int i) {
        // "OK" button was clicked
      }
    })
    .show();
```

Cách dùng này sẽ thực hiện từng bước một, mỗi bước giúp ta khởi tạo được một phần cụ thể của đối tượng AlertDialog.

Đoạn code trên sẽ tạo ra một dialog như hình dưới:



Dependency Injection

Dependency Injection giống như việc di chuyển vào một ngôi nhà đã có đầy đủ đồ đạc nội thất. Ta không cần chuẩn bị mà chỉ việc dùng luônnhững gì cái sẵn có.

Nó cung cấp một đối tượng mà bạn yêu cầu khi bạn muốn khởi tạo nhanh một đối tượng mới, các đối tượng mới không cần xây dựng hay tuỳ chỉnh nhữngthành phần con của nó.

Trong Android, đôi khi ta cần truy cập đến một đối tượng phức tạp từ nhiều điểm trong ứng dụng, ví dụ như: *network client*, *image loader*, SharedPreferences *hay local storage*. Ta có thể inject những đối tượng đó vào *Activity* hoặc *Fragment* và truy cập chúng ngay lập tức khi cần.

Dagger 2 là một open-source dependency injection frameworkthường được sử dụng trong Android, được phát triển bởi sự cộng tác giữa Google và Square. Với *Dagger 2*, bạn có thể chú thích đơn giản một class với @Module annotation và dần bổ sung các method với @Provides annotation như ví dụdưới đây:

```
public class AppModule {
    @Provides SharedPreferences provideSharedPreferences(Application
app) {
        return app.getSharedPreferences("prefs",
Context.MODE_PRIVATE);
    }
}
```

Các module bên trên sẽ tạo và cấu hình tất cả các đối tượng cần thiết. Nó sẽ giúp ích trong quá trình xây dựng các ứng dụng lớn, ta có thểtạo các module tuỳ theo từng chức năng.

Sau đó, ta tạo Component interface để liệt kê các module và class mà ta sẽ inject.

```
@Component(modules = AppModule.class)
interface AppComponent {
    ...
}
```

Cuối cùng, ta sử dụng @Injectannotation để request đến các dependency mỗi khi cần đến nó.

@Inject SharedPreferences sharedPreferences;

Như ví dụ trên, cách tiếp cận này trong Activity sẽ giúp đơngiản việc sử dụng SharedPreferences để lưu trữ cục bộ, mà Activity không cần biết SharedPreference sẽ được khởi tạo và chuyển đến như thế nào.

Bạn có thể tìm hiểu thêm về Dagger theo link: https://google.github.io/dagger/

Singleton

Singleton Design Patter chỉ định rằng sẽ chỉ tồn tại một instance duy nhất của một class nào đó trong toàn bộ chương trình. Nó được dùngkhi ta muốn mô hình hoá một đối tượng cụ thể trong thế giới thực và chỉ có một instance duy nhất.

```
public class ExampleSingleton {
  private static ExampleSingleton instance = null;
```

```
  private ExampleSingleton() {
    // customize if needed
  }
  public static ExampleSingleton getInstance() {
    if (instance == null) {
      instance = new ExampleSingleton();
    }
    return instance;
  }
}
```

Trong đoạn code trên, ở method getInstance sẽ đảm bảo rằng bạnchỉ khởi tạo đối tượng của class một lần duy nhất. Để truy cập đến singleton bạn chỉ cận gọi:

```
ExampleSingleton.getInstance();
```

Structural Patterns

Adapter

Adapter design pattern cho phép 2 lớp không tương thích làmviệc được cùng nhau bằng cách chuyển đổi giao diện (interface) của một lớp sang giao diện khác phù hợp với yêu cầu của khách hàng.

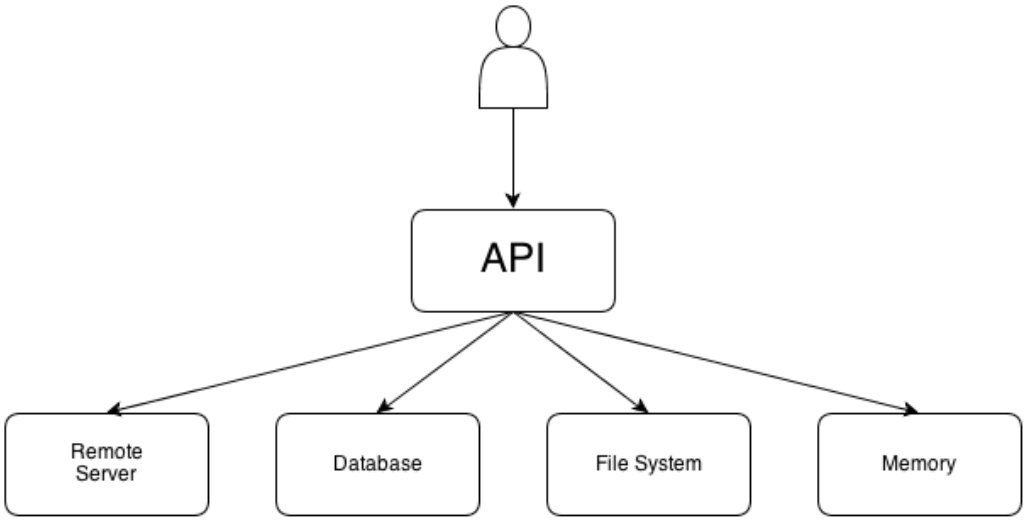
Ví dụ khi ta muốn hiển thị một danh sách các đối tượng cho tương thích với view.

```
public class TribbleAdapter extends RecyclerView.Adapter {
  private List mTribbles;
  public TribbleAdapter(List tribbles) {
    this.mTribbles = tribbles;
  }
  @Override
  public TribbleViewHolder onCreateViewHolder(ViewGroup viewGroup,
int i) {
    LayoutInflater inflater =
LayoutInflater.from(viewGroup.getContext());
    View view = inflater.inflate(R.layout.row_tribble, viewGroup,
false);
    return new TribbleViewHolder(view);
  }
  @Override
  public void onBindViewHolder(TribbleViewHolder viewHolder, int i)
{
    viewHolder.configure(mTribbles.get(i));
  }
  @Override
  public int getItemCount() {
    return mTribbles.size();
  }
}
```

RecyclerView sẽ không biết Tribble là gì, vì nó không bao giờthấy cụ thể, đó là công việc của Adapter. Adapter sẽ có nhiệm vụ xử lý dữ liệu trong Tribble và gửi cấu hình chính xác đến ViewHolder.

Facade

Façade design pattern cung cấp các high level interface để các interface khác dễ sử dụng hơn.



Nếu Activity của bạn cần một danh sách Book, nó có thể yêu cầu một đối tượng duy nhất cho danh sách đó mà không hiểu các hoạt động bên trong như lưu trữ cục bộ, bộ nhớ cache và các API. Nó sẽ giữ cho các đoạn mã được rõ ràng, sạch sẽ mà không làm thay đổi hoạt động trong Activity. Sau này khi cần, ta có thể thay đổi các API mà không làm ảnh hưởng đến các tác động mà nó thực hiện.

Retrofit là một Open Source của Square sẽ giúp ta thực thi Façade design pattern.

Client chỉ đơn giản gọi listBook() method để nhận được danh sách các Book trong callback.

Điều này cho phép ta thực hiện các điều chỉnh bên dưới mà không ảnh hưởng đến client.

Ví dụ: bạn có thể chỉ định deserializer JSON tùy chỉnh:

```
RestAdapter restAdapter =
  new RestAdapter.Builder()
      .setConverter(new MyCustomGsonConverter(new Gson()))
      .setEndpoint("http://www.myexampleurl.com")
      .build();
return restAdapter.create(BooksApi.class);
```

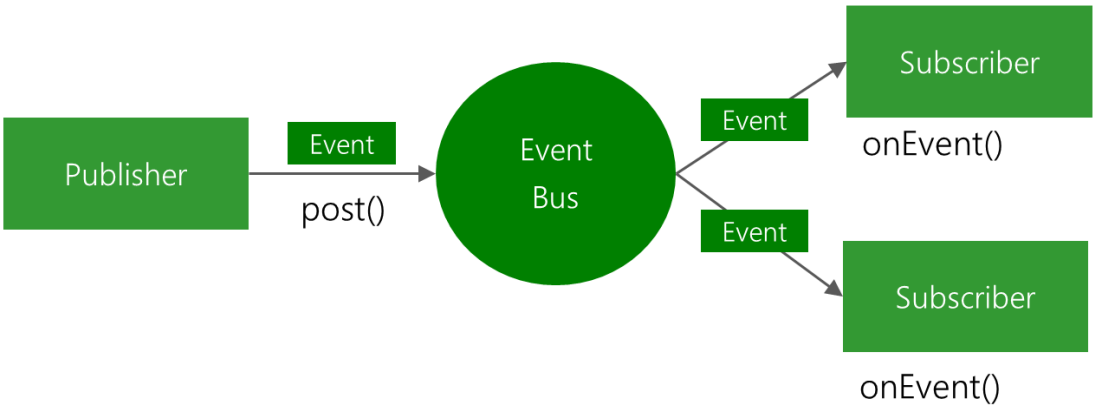
Mỗi đối tượng càng ít biết về những gì diễn ra sau đó (ít phụ thuộc lẫn nhau) thì ta sẽcàng dễ trong việc sửa và điều chỉnh ứng dụng sau này.

Behavioral Patterns

Command

Command design pattern giúp ta phát hành đi các request mà không cần biết người nhận.Ta đóng gói các yêu cầu như là các đối tượng và gửi đi. Việc thực hiện yêu cầu như thế nào sẽ ở một cơ chế khác và không quan tâm ở đây.

EventBus của Greenrobot là một thư viện opensource phổ biến hỗ trợ design pattern này.



Một Event là một đối tượng kiểu lệnh (command-style) được kích hoạt bởi user input, server data hay bất cứ thành phần nào trong ứng dụng. Ta có thể tạo các lớp controng đó mang dữ liệu:

```
public class MySpecificEvent { /* Additional fields if needed */ }
```

Sau khi định nghĩa sự kiện, ta sẽ có được một instance của EventBus và đăng ký một đối tượng như một subscriber.

```
eventBus.register(this);
```

Giờ đây đối tượng đã là một subscriber, ta sẽ nói cho nó biết loại của event và định nghĩa những hành động khi nhận được event đó.

```
public void onEvent(MySpecificEvent event) {/* Do something */};
```

Cuối cùng, tạo và đăng một trong các sự kiện đó dựa trên tiêu chí của mình.

```
eventBus.post(event);
Observer
```

Observer design pattern xác định một phụ thuộc một – nhiều giữa các đối tượng. Khi một đối tượng thay đổi trạng thái, các đối tượng phụ thuộc nó sẽ được thông báo và tự động cập nhật.

Đây là một mô hình linh hoạt, ta có thể sử dụng nó cho các hoạt động có thời gian không xác định, ví dụ như gọi API. Ta cũng có thể sử dụng nó để phản hồi với input từ người dùng.

RxAndroid Framework sẽ dẫn ta thực hiện pattern này xuyên suốt ứng dụng.

```
apiService.getData(someData)
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe (/* an Observer */);
```

Trong một thời gian ngắn, ta xác định các đối tượng Observable sẽ phát giá trị (thông báo). Các giá trị có thể phát cùng lúc, như một luồng liên tục, ở bất cứ tốc độ và thời gian nào.

Các đối tượng Subscriber sẽ lắng nghe những giá trị trên và phán ứng khi chúng đến. Ví dụ, bạn có thể mở đăng ký (subscribe) khi gọi API, lắng nghe response từ server và thực hiện hành vi tương ứng.

Model-View-Controller

MVC đề cập đến mô hình kiến trúc đang thống trị trên nhiều nền tảng. Nó đề cập đến sự phân chia các lớp thành 3 loại:

- *Model*: Các lớp đại diện cho dữ liệu. nó là những mô hình cho thế giới thật.
- *View*: Các lớp trực quan, đang đương trong việc hiển thị cho người dùng.
- *Controller*: Là trung gian của 2 loại trên. Nó cập nhật *View*, lấy input từ người dùng và thực hiện những thay đổi trong *Model*.

Model-View-ViewModel

Một kiến trúc khá giống với MVC. Hai thành phần *Model* và *View* giống với MVC. Thành phần *ViewModel* là trung gian giữa *View* và *Model*, nhưng hoạt động hơi khác với *Controller*. Thay vào đó nó sẽ cung cấp lệnh cho *View* và bind chúng với *Model*. Khi *Model* cập nhật, các *View* tương ứng cũng sẽ được cập nhật từ các ràng buộc dữ liệu với *Model*. Tương tự khi người dùng tương tác với *View*, các ràng buộc sẽ hoạt động theo chiều hướng ngược lại và sẽ cập nhật lên *Model*. Với mô hình này ta sẽ loại bỏ được khá nhiều code trung gian để kết nối giữa *Model* và *View*.

Kết luận

Trên đây là một số Design Pattern thường được sử dụng trong Android mà tôi biết. Nếu bạn có bất kỳ thắc mắc hoặc ý kiến gì vui lòng like và comment bên dưới để chúng ta cùng trao đổi.

Tài nguyên tham khảo:

- Sách: Head First Design Pattern
- https://blogs.agilefaqs.com
- https://viblo.asia
- https://toidicode.com
- https://medium.com
- https://www.codementor.io/@kevinkononenko/model-view-controller-mvc-explained-through-ordering-drinks-at-the-bar-i7nupj4oe
- https://allaravel.com/blog/design-pattern-su-tien-hoa-trong-lap-trinh