Clean Code

Contents

Như thế nào là Clean.

Code phải được thực thi được, Code phải thường xuyên được tối ưu. Code có thể test được và test thành công,

Có hiệu quả:

Tạo hàm để tối sử dụng lại, tránh sử dụng if else, switch mà phải sử dụng tính đa hình hoặc sử dụng factory design để thay thế xem ví dụ bên dưới.

Example:

Khi đi trong 1 cửa hàng bánh, trong task làm bánh thì tùy vào từng loại bánh mà ta làm các công đoạn khác nhau.

```
// bad way
public void doingTaskBad(String type) {
    if ("Biscuit".equals(type)) {
        // do somethings
        System.out.println("Biscuit to doing");
    } else if ("Bread".equals(type)) {
        // do somethings
        System.out.println("Bread to doing");
    } else {
        // nothing
    }
}

// good way use interface instead
public void doingTaskGood(IBakery bakery) {
    bakery.doingBakary();
}
```

Không nên sử dụng như thế vì sau này khi cần thêm loại bánh trả về sẽ phải thay đổi source vi phạm nguyên tắc OCP (Open Closed Principle) ta phải sử dụng interface để nhận vào đối số như sau:

Tạo hàm doingBakary() trong interface.

```
1 lated problem
public interface IBakery {
    void doingBakary();
}
```

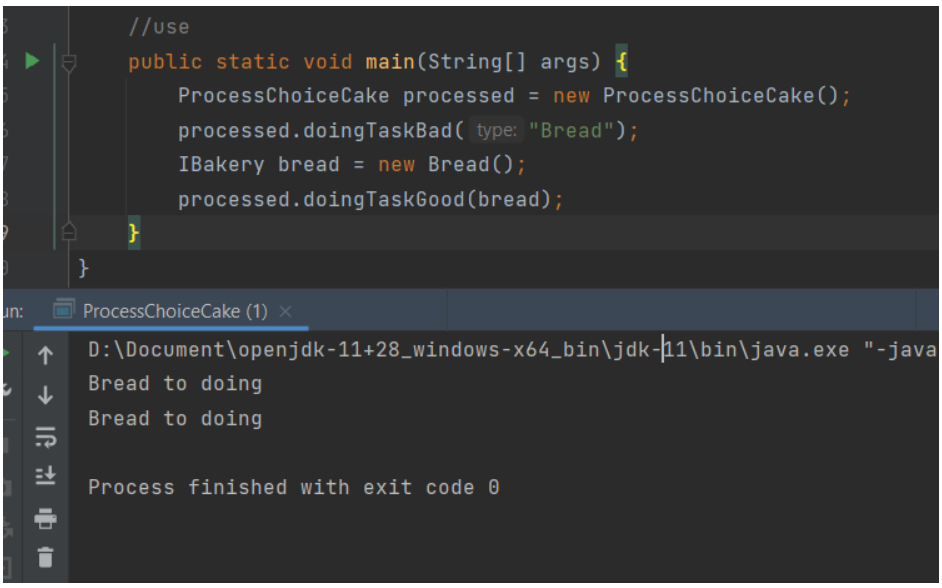
Bread kế thừa từ inteface và implement hàm doingBakary() để hành động theo riêng cách của mình.

```java
public class Biscuit implements IBakery{
    @Override
    public void doingBakary() {
        System.out.println("Biscuit to doing");
    }
}
```

```java
public class Bread implements IBakery{
    @Override
    public void doingBakary() {
        System.out.println("Bread to doing");
    }
}
```

Use

The result of bad and good structure same.



Có thể đọc được,

Tránh những class phụ thuộc vào nhau quá. Khi có bug thì sẽ ảnh hưởng dây chuyền. Code có thể bảo trì , có thể extend được.

Chính bản thân của code sẽ là document. isEmpty, isNotNull, getDataAddressCustomer,..

Bad code là gì

Những comment lỗi thời: có nghĩa là code đã thay đổi nhưng comment vẫn giữ lại. Comment không khớp với code.

Comment dư thừa: không đáng để comment.

Viết comment không hiểu: không đầu tư vào comment viết thiếu chỉnh chu

Không cần comment outcode: kiểu như khi viết code replace code cũ rem code cũ, không nên rem xóa thẳng luôn.

Thông thường function tối đa 3 thông số input thôi.

Tránh sử dụng cờ.

Những hàm chết nên bỏ đi.

Duplication code: code trùng ở nhiều chỗ

Xử lí không chính xác ở giá trị biên: niên kiểm tra ở giá trị biên

Không đặc đúng vị trí của level abstract: đặc sai vi trí hàm hoặc property.

Lớp base không được phụ thuộc lớp cụ thể: kiểu như ở level lớp cha thì không phụ thuộc vào level lớp con.

Quang Trọng của clean code

Giảm chi phí tranning cho người đi sau.

Nhanh chóng maintain tốt.

Improve Clean code

Đăc tên tốt cho biến, hàm, class, interface, package, project.

Biến : bắt đầu bằng chữ thường, đặt theo danh từ ex: total, report,..

Hàm: bắt đầu bằng chữ thường đúng theo nhiệm vụ của nó, đừng gại đặc tên hàm dài đặt theo Verb động từ, calculatorTotal,...

Khi viết hàm tránh tách dụng phụ làm đúng nhiệm vụ của mình: giống như hàm này xài thì call luôn những nhiệm vụ khác.

Class: bắt đầu bằng chứ Hoa, đặt theo danh từ: Employee,...

Interface: giống với các đăc tên class, nên thêm I trước là Iperson,...

Tránh đặc tên gây hiểu nhầm, hoặc không có nghĩa: s1, s2, abc,..

Tên dễ gợi nhớ.

Contructor overloaded ưu tiên dùng factorymethod (cách dùng of sử dụng của java): bởi vì factory sẽ ủy chuyển hơn có thể trả về 1 loại khác hoặc xử lí thêm để đưa ra 1 object.

```java
// example class tree use constructor overloaded
public class Tree {
    private String name;
    private String type;
    private String root;
    private String body;
    private String bough; // cành cây
    private String leaf; // lá cây

    public Tree() {}
    private Tree(String name) { this.name = name; }
    private Tree(String name, String type) {...}
    private Tree(String name, String type, String root) {...}
    private Tree(Tree other) {...}

    public static Tree of(String name) {...}
    public static Tree of(String name, String type) {...}
    public static Tree of(String name, String type, String root) {...}
    public static Tree from(Tree otherTree) {...}

}
```

Đặc nên tên nhất quán: khiểu như get thì get hết chứ không lúc get, lúc fetch, lúc retrive.

Khuyến khích đặt tên hàm theo Technique. Ex: accountVistor có nghĩa là account và vistor là vistor pattern.

Kỹ thuật Refactor Code

Extract method: nếu code phức tạp và nhiều chỗ dùng nó có thê giải ra thành 1 hàm.

Inline method: ngược lại extract method, nhìn vô là biết rồi không cần tạo hàm.

Extract variable: tạo ra biến mới cho dòng nếu nó phức tạm quá. Kiểu như trong condition của if phức tạp quá thì tạo biến cho nó.

Bad Code

```java
public static void main(String[] args) {
    int widthSize = 50;
    int heightSize = 100;

    if((widthSize + 10 - 8 * 8 / heightSize) == 90){
        // do something
    }else {
        // do something
    }
}
}
```

Nên tách nó thành variable

```java
public static void main(String[] args) {
    int widthSize = 50;
    int heightSize = 100;

    int total = widthSize + 10 - 8 * 8 / heightSize;
    if(total == 90){
        // do something
    }else {
        // do something
    }
}
}
```

Sliptemporaryvariable: 1 biến không nên mang nhiều giá trị , ví dụ biến temp dán cho nhiều giá trị là không nên, không cần tiết kiệm bộ nhớ lúc này.

Remove assignment from parameter: không nên thay đổi giá trị input truyền vào, gây khó cho người đọc code. Solution tạo ra biết khác trả về.

```java
//bad
void takeCareOfPlant(Tree tree){
    if("n".equals(tree.getTakeCareStatus())){
        tree.setTakeCareStatus("y");
    }
}
//good
Tree takeCareOfPlantGood(Tree tree) throws CloneNotSupportedException {
    Tree treeClone = tree.clone();
    if("n".equals(tree.getTakeCareStatus())){
        treeClone.setTakeCareStatus("y");
    }
    return treeClone;
}
public static void main(String[] args) throws CloneNotSupportedException {
    UnChangeValueInputOfMethod unChangeValueInputOfMethod = new UnChangeValueInputOfMethod();
    Tree treeInSideChange = new Tree();
    Tree treeChangeFromCloneObject  = new Tree();
    unChangeValueInputOfMethod.takeCareOfPlant(treeInSideChange);
    unChangeValueInputOfMethod.takeCareOfPlantGood(treeChangeFromCloneObject);
    System.out.println(treeInSideChange.toString());
    System.out.println(treeChangeFromCloneObject.toString());
}
```

```
Run: UnChangeValueInputOfMethod
D:\Document\openjdk-11+28_windows-x64_bin\jdk-11\bin\java.exe "-javaagent:D:\Tool\IntelliJ IDEA Commur
Tree{takeCareStatus='y'}
Tree{takeCareStatus='n'}
```

Replace method with object: tách method phức tạp thành 1 class mới.

Các funtion và field trong Object

Moving feature and fields between object: do design sai, cần phải đưa fields, hàm về đúng vị trí nó.

Extract class: đem những thứ dư thừa thành 1 class riêng biệt không để nhiều thứ trong 1 class quá.

Hide deletegate: Call trực tiếp không cần phải thông qua 1 object trung giang

VD: Class lớp học chứa Class Giáo viên nhiệm, Class học sinh chứa Class Lớp học. Yêu cầu từ Class học sinh muốn get Giáo viên Chủ nhiệm.

Bad Code: Từ học sinh get Lớp học -> Giáo viên chủ nhiệm

Good Code: tạo ra 1 hàm để làm việc trên, mỗi lần muốn get call function đó.

```
Teacher teacher = new Teacher();
teacher.setName("teacher lucy");

ClassRoom room = new ClassRoom();
room.setName("101");
room.setMainTecher(teacher);

Student student = new Student();
student.setName("John");
student.setRoom(room);

// how to get teach of student John

//bad
ClassRoom roomRetrieve = student.getRoom();
roomRetrieve.getMainTecher();

//Good create function in student to get teacher main
student.getTecherMain();
```

Introduce foreign method: 1 hàm dài quá thì tách ra thành nhiều function liên kết với nhau.

Introduce local extention: nếu hàm này có nhiều chỗ sử dụng thì tạo ra 1 class Util để nhiều chỗ được sử dụng .

Cách thức tổ chức

Self encapsulated: nghĩa là những hàm get set thì ta nên sử dụng get set thay vì sử dụng trực tiếp, lỡ trong get set có cái if else lọc điều kiện đảm bảo an toàn cho property.

Replace data with object: thay đổi properties của class thành object thay vì 1 datatype.

VD: Có class **hocsinh** và **classroom**. Và yêu cầu là muốn lưu classroom trong hocsinh.

Bad code: class hocsinh { private Long classroom; } chỉ lưu mã lớp học

Bad code: class hocsinh { private Classroom classroom; } lưu cả object lớp học

Khi cần sử dụng thì tiện lợi hơn. Ví dụ như muốn lấy Giáo viên chủ nhiệm ta get từ classroom.getGVCN().

Change value to reference: khi sử dụng chung 1 đối tượng nào đó thì không nên sử dụng new contructor mà ta sử dụng 1 factorymethod để get ra, kểu như single ton.

Ví dụ: Trong **học sinh** cần lưu đối tượng **lớp học.**

Bad: Tạo ra 20 học sinh -> phải tạo 20 lớp học để lưu vào học sinh.

Good: Tạo ra static list lớp học trong class lớp học.

Tạo ra 20 học sinh -> kiểm tra lớp học tồn tại trong list không nếu tồn tại lấy ra không tạo mới, nếu không có thêm vào list.

Như vậy 20 học sinh -> chỉ có 1 class lớp học được tạo tiết kiệm bộ nhớ.

Change reference to value: giống như ta muốn tạo ra 1 object contain value của nó, không tạo ra hàm set, và các property của nó cũng là các class Wapper immutable. Khởi tạo class như thế nào thì xài thế đấy không change được giá trị trong nó. Lưu ý là không tồn tại hàm set nào trong class.

Change array to class: dùng array quản lí đối tượng thì không tối ưu lắm, nên chuyển về class thay cho array. Ví dụ những tường hợp nào thấy class lưu cấu trúc kì quá thì ta sử dụng object thay thế.

Duplication observed data: Hạn chế làm những việc giống như là việc sử dụng observer pattern.

Vd: Ta có school, student và teacher. Yêu cầu là shool cần thông báo 1 lần cho tất cả các student và teacher của trường.

Bad code:

code school -> thông báo -> cho student.

code school -> thông báo -> cho teacher.

Good code: Áp dụng Observerable và observer

Observerable(School) -> thông báo -> observer(include student and teacher).

Change unidirectional association to bidiretional: cho quan hệ của 2 class 2 chiều quan hệ với nhau, trong book có author, trong author có danh sách các book.

Change bidiretional associate to unidirectional: chuyển về quang hệ 1 chiều, chỉ có author chứa ds cách book. Còn book không biết author.

Symbolic constant: dùng constant không dùng hard code.

Encapsulation collection: tránh trả về địa chỉ của collection, người dùng có thể change các item trong collection đó làm thay đổi phá vỡ tính đóng gói của OOP. Trong trường hợp này có thể create 1 unModifiableCollection read only (ở đây trả về 1 collection không edit, hoặc thêm mới, nhưng vẫn có thể edit các item của collection modify conten of item). Còn đối với object thì có thể dùng Clone object đó. Muốn thay đổi các item thì phải định nghĩa trong chính class đang chứa collection để đảm bảo an toàn không thay đổi collection đó từ bên ngoài.

More constant to enum: sử dụng enum thay thế cho các constant.

Replace type code with subclass: chia class thành các nhiều subclass nếu quá phức tạp

Nguyên lý code của OOP.

Design Class:

SRP (Single reposibility Principle): Một class chỉ có một duy nhất 1 nhiệm vụ không làm da nhiệm vụ

OCP (Open Closed Principle): Bạn có thể extend class, nhưng không modify nó. Trong trường hợp thêm feature mới thì cần extend chứ không thay đổi code cũ để nó run được. Kiểu như sử dụng Astracfactory

LSP (Liskov Substitution Princple): Phương thức một lớp con phải có ý nghĩa tương đồng với hành động lớp cha chứ không được khác ý nghĩa.

DIP (Dependency Inversion Principle): high level detail không phụ thuộc vào low lever. Nên sử dụng interface để tránh sử dụng low level.

ISP (Interface Segregation Principle): khi inplement interface có những hàm bỏ trống, là hành vi phạm nguyên tắc này. Cách tránh thì ta tạo thêm 1 interface mới phù hợp với tình huống hiện tại.

Package Design:

REP (The Reuse Equivalence Principle): reuse được có nghĩa khi sử dụng không cần main nó, không cần phải sử lại code đó, khi nâng cấp version không bắt buộc phải thay đổi nâng cấp nó.

CCP (Common Clusure Priciple): Những class change thì change cùng nhau thì nên được đặt cùng chung package. Những class sử dụng nhau thì nên chung package.

IDP (Acyclic Dependency Principle): Không được design 2 quan hệ cắn đuôi nhau kiểu như quang hệ xoay vòng khi có bug sẽ xảy ra liên hoàng bug. Solution là sẽ tạo lớp trung giang.

Reference:

https://blog.devgenius.io/clean-code-b1fed462faa8

https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29