Technical Review

junyan.xu[*]

January 1, 2019

Contents

[*]junyan.xu@jpmorgan.com

1 SOLID Principles

Solid principles are the most basic principles in system design

1.1 Single Responsible principle

Suppose you are going to design two classes, *ractangle* and *triangle*.

```
class shape{
private:
        int width;
        int height;
};

class rectangle: shape {

};

class triangle: shape{
};
```

Then you want to have a factory function to calculate the size. The function itself should only return the size. But you are not satisfied with it, you want to return an object with other functions.

```
struct summary{
        int size;
        string format;
        string to_string(){
                if(format == ".2d")
                        return ....
        }
};

struct getSize(const shape* s, const string& format){
}
```

However, this is not a good design since it violate the single responsibility principle. Since now getSize is not only calculate the size, but also in charge of formatting the output. Which add extra interconnection and logic to the code. So we need to refactor the code into following

```
struct summary{
        int size;
        string format;
        string to_string(){
                if(format == ".2d")
                        return ....
        }
};

summary getFormatObject(int size, const string& format){
}

struct getSize(const shape* s){
}
```

The single responsible principle tell us to write a function only for a single purpose. Do not control to many things.

1.2 Open/Close principle

Inside the get size function, a short cut. We can write the function with bunch of if-else:

```
struct getSize(const shape* s){

        // dynamic_cast here can check the actual type safely
        // at the run time

        if ( dynamic_cast<rectangle*>( s ) )
        // do something

        else if ( dynamic_cast<triangle*>( s ) )
        // do something else
        }
}
```

The function above is not a good way of code because it violate the open/close principle. If you have another class circle, then you will have to **modify** the function method. **open** in the principle means open for extension (add more method) while **close** here means close for modification of the old method.

```
class circle: shape {

};

int getSize(const shape* s){

        if ( dynamic_cast<rectangle*>( s ) ){
        }
        else if ( dynamic_cast<triangle*>( s ) )
        }
        else if ( dynamic_cast<circle*>( s ) )
        // modify the getSize every time you add new class features
        }
}
```

Correct way of coding is add getSize method as a virtual member function of base and derived class

```
class shape{
private:
        int width;
        int height;
public:
        virtual int getSize() const;
};

class rectangle: shape {
        int getSize() const{
        }
};

class triangle: shape {
        int getSize() const{
        }
};

class circle: shape {
        int getSize() const{
        }
};

int getSize(const shape* s){

        return s->getSize();
}
```

Then every method is closed. If we want to have new class, no existing code section will be modified.

1.3 Liskov Substitution principle

Liskov substitution principle is about how to design base and child to be inter-changeable. The famous eclipse-circle problem is

```
class eclipse{
private:
        float a;
        float b;
public:
        eclipse(float a, float b): a(a), b(b){}
}
```

```
class circle: eclipse{
public:
        circle(float r): eclipse(r, r){}
private:
}
```

Some people would say it is a good design. But, to Liskov principle tell us that the child class must provide everything that parent class method. Think about if our eclipse class have a method **stretch**

```
class eclipse{
private:
        float a;
        float b;
public:
        eclipse(float a, float b): a(a), b(b){}
        stretch(float a, float b){this-> a = a; this->b = b;}
}
```

Then our circle must have this method too. However, circle cannot have this method. So here are two possible solutions:

- throw exception or return failure value in circle stretch method
- redefine the data model, add another common base class "shape" to both of it. Now circle vs eclipse is D vs D (Derived vs Derived)

1.4 Interface Segregation Principle

In a short word, base interface should not have too much functions, which means the derived class should never be forced to have the method it does not use.

```
class shape{
private:
        int width;
        int height;
public:
        virtual int getSize() const;
        virtual int getVolume
};

class rectangle: shape {
/*
        rectangle here is forced to implement a getVolume.
*/
        int getSize() const{
        }
};


class cubic: shape {
        int getVolume const{
        }
};
```

A good solution is bifurcate the original shape interface into two parallel sub interface class. Then we segregate methods that have conflict.

```
class BaseShape{
private:
        int width;
        int height;
};

class FlatShape: BaseShape{
/*
        segregate the getVolume and getSize into two branch
*/
        virtual int getSize const(){
        }
}

class SolidShape: BaseShape{
        virtual int getVolume const(){
        }
}

class rectangle: FlatShape {
        int getSize() const{
        }
};

class cubic: SolidShape {
        int getVolume const{
        }
};
```

1.5 Dependency Inversion Principle

This is the easiest of five. The higher level function should only rely on the lower level abstractions but not the detailed implementation. The bridge pattern, abstract factory pattern are good example. Here the high-level user paint depend on the abstraction of different Painter, no matter how painter is changed. The rectangle can always get the updated result.

Adding abstraction dependency is called dependency injection.

```
class Painter:{
public:
        virtual void paint(){}
};

class FancyPainter: Painter{
public:
        void paint(){
        // fancy painting
        }
};

class rectangle: shape{

// high level user rectangle only need to know we have a member
// which is in charge of paint. That's it, no more info that high-level
// user need to know.

public:
        rectangle(float x, float y, Painter * pt): width(x), width(y), pt(pt){
                // add Painter* here is actually called dependency injection
        }

        void paint(){
                pt.paint();
        }
private:
        Painter pt;
};
```

2 Management Type Object Design

3 Reservation Type Object Design

4 Card Game Type Object Design