MỘT SỐ PATTERN THAM KHẢO CHO PHÁT TRIỂN HỆ BoEC

- Fctory
- Builder
- Adapter
- Façade

Giới thiệu về Factory method pattern.

https://viblo.asia/p/factory-method-pattern-trong-java-4dbZNoNQlYM

Factory method là một pattern cho việc khởi tạo đối tượng(thuộc nhóm creational patterns). Pattern này được sinh ra nhằm mục đích khởi tạo đối tượng mà bản thân muốn che giấu class nào được khởi tạo. Pattern này được sử dụng khá phổ biến đồng thời nó cũng không khó khăn để hiểu.

Bản chất của Factory method pattern

Về cơ bản thì ta sẽ định nghĩa một `interface` hoặc `Abstract class` , các class con sẽ `implements` nó. Tiếp đến mình sẽ tạo một class được xem là **FactoryClass**, bên trong **FactoryClass** này mình sẽ có một phương thức giúp chúng ta khởi tạo các Object chúng ta cần.

Code demo

Mình chọn điện thoại làm một cái ví dụ ha.

- Như mình viết trên thì mính sẽ tạo một `interface Phone`

```
public interface Phone {

    public void showInfo();

}
```

- Tiếp đến mình muốn có một điện thoại của một vài hãng như Samsung, Apple, Nokia,.. Nên mình sẽ khai báo các class SamsungPhone, ApplePhone, NokiaPhone

```
public class SamsungPhone implements Phone {

    @Override
    public void showInfo() {
          System.out.printf("Đây là điện thoại Samsung");
    }

}
public class ApplePhone implements Phone {

  @Override
  public void showInfo() {
        System.out.printf("Đây là điện thoại Apple");
  }

}
public class NokiaPhone implements Phone {

  @Override
  public void showInfo() {
        System.out.printf("Đây là điện thoại Nokia");
  }
```

```
•
• }
• public enum PhoneType {
•   SAMSUNG, NOKIA, APPLE
• }
```

Sau khi mình có được các class *Phone trên thì sẽ tạo `FactoryClass` có tên là `PhoneFactory`

```
public class PhoneFactory {

   public Phone getPhone(PhoneType phoneType) {

       Phone phoneCreated = null;

       switch (phoneType) {

       case SAMSUNG:

           phoneCreated = new SamsungPhone();

           break;

       case APPLE:

           phoneCreated = new ApplePhone();

           break;

       case NOKIA:

           phoneCreated = new NokiaPhone();

           break;

       }

       return phoneCreated;

   }
```

```
}
```

Bây giờ mình sẽ chạy demo

```
public class RunDemo {


   public static void main(String[] args) {
       PhoneFactory phoneFactory = new PhoneFactory();
       Phone phone = phoneFactory.getPhone(PhoneType.SAMSUNG);
       phone.showInfo();
   }
}
```

Và kết quả có được là;

```
Samsung phone
```

Như vậy khi muộn tạo object ta sẽ dùng class PhoneFactory và chỉ loại object cần tạo để Factory method sẽ tạo object cần thiết cho bạn.

Áp Dụng Builder Pattern

http://ktmt.github.io/blog/2013/06/14/design-pattern-ap-dung-builder-pattern-trong-test-java/

Giới thiệu

Builder Pattern là một pattern trong cuốn Design Pattern của “Gang of Four”. Mục đích của Builder Pattern như sau: Separate the construction of a complex object from its representation so that the same construction process can create different representations (tạm dịch: tách rời quá trình tạo object với nội dung và cấu trúc bên trong của nó, nhờ vậy tương ứng với một quá trình tạo object có thể có nhiều cách tạo nhiều thể hiện khác nhau.)

Bên cạnh đó, Builder pattern còn được dùng để giải quyết một vấn đề thường gặp trong quá trình test: cách nào tốt nhất để tạo những object có quá nhiều optional parameters, trong khi tạm thời một số parameter ta muốn để giá trị default, giá trị của parameter này không ảnh hưởng gì lắm đến algorithm hay flow của chương trình. Bài viết này sẽ trình bày phương pháp sử dụng Builder pattern này.

Ví dụ

Ta lấy ví dụ ta có một class quản lý sách, với các trường như sau: tiêu đề, tên tác giả, thể loại, năm xuất bản, ISBN. Để phục vụ việc tạo một object thuộc class này, ta nghĩ đến việc tạo một constructor như sau:

constructor.java

```
1   public Book(String title, String author, Genre genre, GregorianCalendar
2  publishDate, String ISBN)
3    {
4        this.title = title;
5        this.author = author;
6        this.genre = genre;
7        this.publishDate = publishDate;
8        this.ISBN = ISBN;
    }
```

Và trong chương trình, để tạo một object Book, ta gọi:

constructor.java

```
1  Book book2 = new Book("Core Java", "Cay Horstman", Genre.TECHNOLOGY, new
   GregorianCalendar(2012,12,7), "0137081898");
```

Ta thấy, với cách tạo object như trên, có một số nhược điểm như sau:

- Ta bắt buộc phải khai báo tất cả các parameters, không có giá trị default. Ta sẽ bị buộc phải set cả những parameters mà ta không quan tâm khi thực hiện test object.
- Đoạn code khá khó hiểu khi ta chỉ khai báo giá trị của parameter và truyền vào constructor. Người đọc sẽ phải đếm vị trí của parameters, soi vào trong khai báo constructor của Book.java để biết giá trị này là gán cho parameter nào. Với ví dụ trên, chỉ có 5 parameter, nhưng với những class phức tạp có nhiều parameters hơn nữa, việc khai báo như trên rõ ràng không tốt về mặt code visibility.
- Xuất hiện nhu cầu tạo object với các constructor với bộ tham số đầu vào khác nhau. Như ví dụ ở trên, ta muốn tạo object nhưng không muốn nhập thể loại, hoặc không muốn nhập năm xuất bản, … dẫn đến rất nhiều phiên bản constructor khác nhau.

Hoặc bạn có thể khai báo theo một cách thứ hai, theo kiểu JavaBean như sau: tạo một constructor default, không đối số, ví dụ như Book(), sau đó thì tạo một loạt các setter để nhập các tham số cho parameter. Lúc đó thì code cho từng đối tượng sẽ như sau:

bean.java

```
1  Book book = new Book();
2  book.setTitle("Core Java");
3  book.setAuthor("Cay Horstman");
4  book.setGenre(Genre.Technology);
5  book.setPublishDate(new GregorianCalendar(2012,7,1));
6  book.setISBN("0137081898");
```

Nhưng cách này lại có nhược điểm là: vì việc tạo object bị kéo dài qua nhiều câu lệnh, có thể object sẽ bị rơi vào trạng thái unstable (ví dụ như ta quên mất set publishDate, như thế book sẽ không có giá trị cho publishDate !).

Để khắc phục những nhược điểm trên, có một phương pháp sử dụng Builder pattern như sau: Thay vì tạo object mong muốn trực tiếp, ta gọi một static factory với các tham số bắt buộc, nhận về một *builder object*. Sau đó gọi các setter method dể đặt các tham số tùy chọn. Cuối cùng gọi build method, tạo ra object. Để dễ hiểu hơn, ta quan sát ví dụ sau:

builder.java

```
1  public class Book {
2    public enum Genre {FICTION, NONFICTION, TECHNOLOGY, SELFHELP, BUSINESS, SPORT};
```

```
 3
 4    private String title;
 5    private String author;
 6    private Genre genre;
 7    private GregorianCalendar publishDate;
 8    private String ISBN;
 9
10    public static class Builder
11    {
12        //required params
13        private String title;
14        private String author;
15
16        //optional params
17        private Genre genre = Genre.FICTION;
18        private GregorianCalendar publishDate = new GregorianCalendar(1900,1,1);
19        private String ISBN = "000000000";
20
21        public Builder(String title, String author)
22        {
23            this.title = title;
24            this.author = author;
25        }
26
27        public Builder genre (Genre val)
28        {
29            this.genre = val;
30            return this;
31        }
32
33        public Builder publishDate(GregorianCalendar val)
34        {
35            this.publishDate = val;
36            return this;
37        }
38
39        public Builder ISBN(String val)
40        {
41            this.ISBN = val;
42            return this;
43        }
44
45        public Book build()
46        {
47            return new Book(this);
48        }
49    }
50
51    public Book(Builder builder)
52    {
53        title = builder.title;
54        author = builder.author;
55        genre = builder.genre;
56        publishDate = builder.publishDate;
57        ISBN = builder.ISBN;
58    }
59
60    @Override
61    public String toString()
62    {
63        return "Title: " + title + ", author: " + author + ", genre: " +
64  genre.toString() + ", publish year: "
65                    + publishDate.get(Calendar.YEAR) + ", ISBN: " + ISBN;
66    }
67
68  }
69
70
71
72
```

```
67
```

Lúc đó ta có thể tạo Book object bằng cách sau:

builder.java

```
1       Book book = new Book.Builder("Effective Java", "Joshua Bloch")
2                        .publishDate(new GregorianCalendar(2008,05, 28))
3                        .build();
```

Ta thấy, cách tạo object này có ưu điểm hơn so với cách dùng constructor thông thường:

- Không cần phải khai báo những parameter nào mà ta tạm thời chưa quan tâm. Những parameters đó sẽ nhận giá trị default. Trong ví dụ trên, ta đã bỏ qua khai báo cho genre và ISBN. Chúng sẽ nhận giá trị default: genre = FICTION, ISBN = “0000000000”.
- Ta thấy rõ là giá trị nào là gán cho parameter nào. Ví dụ: publishDate(new GregorianCalendar(2008,05, 28)) là gán giá trị cho parameter ngày xuất bản.
- Thứ tự của các method là không quan trọng. Ta có thể gọi các method để update thêm các giá trị cho các parameter theo thứ tự tùy ý. Object sẽ chưa được tạo cho đến khi gọi build(). Do vậy, builder rất dễ sử dụng và giúp ta tránh khỏi những sai lầm không mong muốn đối với thứ tự parameter.

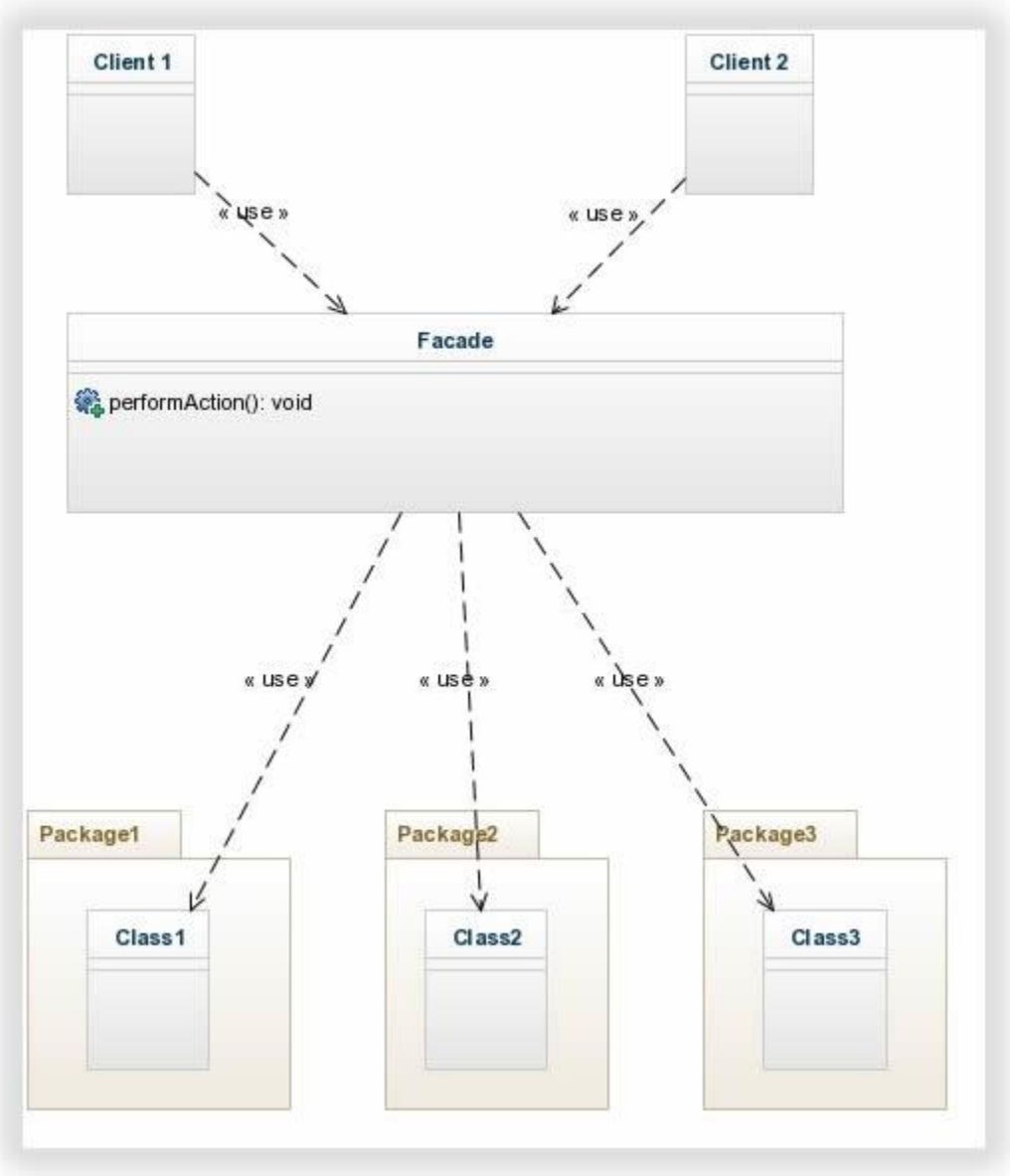
Kết luận

Bài viết đã giới thiệu cách sử dụng Builder Pattern để việc tạo object trong quá trinh test được dễ dàng hơn, đồng thời khiến đoạn code được sáng sủa hơn về mặt trình bày.

FACADE Design Pattern

https://code.tutsplus.com/tutorials/design-patterns-the-facade-pattern--cms-22238

The UML



Problem

Let's assume that you have a few operations to be made in sequence, and that the same action is required in multiple places within your application. You have to place the same code again and again in different places. You have done that, but after a few days you find that something needs to be changed in that code.

Do you see the problem? We have to introduce the changes in all of the places that the code exists. It's painful, isn't it?

Solution

As a solution, what you should be doing is to create a lead controller, which handles all of the repeating code. From the calling point of view, we will just call the lead controller to perform actions based on the parameters provided.

Now if we need to introduce any change in the process, then we will just need to change the lead controller instead of making the change in all places where we used that code.

Example

In this tutorial, let's choose one lesson so that it makes things more readable. Let's say that you have been given a task to plan your friend's marriage. If you do everything on your own, then imagine the things you need to cover. It will create a higher possibility for error, and increase the chance of missing something that can drastically affect your friend's wedding.

In this case, instead of doing everything on your own, you should use a wedding planner and make sure the job gets done in a well-managed manner with less chance of a mistake.

Here, you are behaving as a client who initiates the process, and the wedding planner is working as a "facade" for you, completing the job based on your direction.

Code Example

In this section we will see one more example, which is very common for websites, of course with a code example. We will see an implementation of the facade design pattern using a product checkout process. But before checking perfect code with the facade pattern, let's have a look at some code that has a problem.

A **simple checkout process** includes the following steps:

1. Add product to cart.
2. Calculate shipping charge.
3. Calculate discount.
4. Generate order.

Problem

```
01  // Simple CheckOut Process
02  $productID = $_GET['productId'];
03
04  $qtyCheck = new productQty();
05
06  if($qtyCheck->checkQty($productID) > 0) {
07
08      // Add Product to Cart
09      $addToCart = new addToCart($productID);
10
11      // Calculate Shipping Charge
12      $shipping = new shippingCharge();
13      $shipping->updateCharge();
14
15      // Calculate Discount Based on
16      $discount = new discount();
17      $discount->applyDiscount();
18
19      $order = new order();
20      $order->generateOrder();
21  }
```

In the above code, you will find that the checkout procedure includes various objects that need to be produced in order to complete the checkout operation. Imagine that you have to implement this process in multiple places. If that's the case, it will be problematic when the code needs to be modified. It's better to make those changes in all places at once.

Solution

We will write the same thing with the facade pattern, which makes the same code more maintainable and extendable.

```
class productOrderFacade {

    public $productID = '';

    public function __construct($pID) {
        $this->productID = $pID;
    }

    public function generateOrder() {

        if($this->qtyCheck()) {

            // Add Product to Cart
            $this->addToCart();

            // Calculate Shipping Charge
            $this->calulateShipping();
```

```
17
18              // Calculate Discount if any
19              $this->applyDiscount();
20
21              // Place and confirm Order
22              $this->placeOrder();
23
24          }
25
26      }
27
28      private function addToCart () {
29          /* .. add product to cart ..  */
30      }
31
32      private function qtyCheck() {
33
34          $qty = 'get product quantity from database';
35
36          if($qty > 0) {
37              return true;
38          } else {
39              return true;
40          }
41      }
```

```
42        private function calulateShipping() {
43            $shipping = new shippingCharge();
44            $shipping->calculateCharge();
45        }
46
47        private function applyDiscount() {
48            $discount = new discount();
49            $discount->applyDiscount();
50        }
51
52        private function placeOrder() {
53            $order = new order();
54            $order->generateOrder();
55        }
56    }
57
58
59
```

As of now, we have our product order facade ready. All we have to do is use it with a few communication channels of code, instead of a bunch of code as expressed in the previous part.

Please check the amount of code below which you will need to invest in order to have a checkout process at multiple positions.

```
1  // Note: We should not use direct get values for Database queries to prevent SQL injec
2  $productID = $_GET['productId'];
```

```
3
4   // Just 2 lines of code in all places, instead of a lengthy process everywhere
5   $order = new productOrderFacade($productID);
6   $order->generateOrder();
```

Now imagine when you need to make alterations in your checkout process. Now you simply create changes in the facade class that we have created, rather than introducing changes in every place where it has been applied.

Conclusion

Simply put, we can say that the facade pattern should be carried out in a situation where you need a single interface to complete multiple procedures, as in the example of the wedding planner who is working as a facade for you to complete multiple procedures.

Design Patterns: The Adapter Pattern

https://code.tutsplus.com/tutorials/design-patterns-the-adapter-pattern--cms-22262

In the last article, we looked at how the facade design pattern can be employed to simplify the employment of any large and complex system using only a simple facade class.

In this article, we will continue our discussion on design patterns by taking a look at the **adapter design pattern**. This particular pattern can be used when your code is dependent on some external API, or any other class that is prone to change frequently. This pattern falls under the category of "structural patterns" because it teaches us how our code and our classes should be structured in order to manage and/or extend them easily.

Again, I'd like to reiterate that design patterns have nothing new over traditional classes. Instead, they show us a better way to structure our classes, handle their behavior, and manage their creation.

The Problem

```
<?php
class PayPal {

    public function __construct() {
        // Your Code here //
    }

    public function sendPayment($amount) {
        // Paying via Paypal //
        echo "Paying via PayPal: ". $amount;
    }
}

$paypal = new PayPal();
$paypal->sendPayment('2629');
```

In the above code, you can see that we are utilizing a PayPal class to simply pay the amount. Here, we are directly creating the object of the PayPal class and paying via PayPal. You have this code scattered in multiple places. So we can see that the code is using the `$paypal->sendPayment('amount here');` method to pay.

Some time ago, PayPal changed the API method name from `sendPayment` to `payAmount`. This should clearly indicate a problem for those of

us who have been using the `sendPayment` method. Specifically, we need to change all `sendPayment` method calls to `payAmount`. Imagine the amount of code we need to change and the time we need to spend on testing each of the features once again.

The Solution

One solution to this problem is to use the adapter design pattern.

According to Wikipedia:

> *In software engineering, the adapter pattern is a software design pattern that allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.*

In this case, we should create one wrapper interface which makes this possible. We will not make any changes in the external class library because we do not have control over it and it may change any time.

Let's dig into the code now, which shows the adapter pattern in action:

```
01  // Concrete Implementation of PayPal Class
02  class PayPal {
03
04      public function __construct() {
05          // Your Code here //
06      }
07
      public function sendPayment($amount) {
```

```
08          // Paying via Paypal //
09          echo "Paying via PayPal: ". $amount;
10      }
11  }
12
13  // Simple Interface for each Adapter we create
14  interface paymentAdapter {
15      public function pay($amount);
16  }
17
18  class paypalAdapter implements paymentAdapter {
19
20      private $paypal;
21
22      public function __construct(PayPal $paypal) {
23          $this->paypal = $paypal;
24      }
25
26      public function pay($amount) {
27          $this->paypal->sendPayment($amount);
28      }
29  }
30
```

Study the code above and you should be able to tell that we have not introduced any changes into the main `PayPal` class. Instead we have created one interface for our payment adapter and one adapter class for PayPal.

And so afterward we have made the object of the adapter class instead of the main `PayPal` class. While creating an object of adapter class we will pass the object of the main `PayPal` class as an argument, so that adapter class can have a reference to the main class and it can call the required methods of the main `PayPal` class.

Let's find out how we can utilize this method directly:

```
1  // Client Code
2  $paypal = new paypalAdapter(new PayPal());
3  $paypal->pay('2629');
```

Now imagine PayPal changes its method name from `sendPayment` to payAmount. Then we just need to make changes in `paypalAdapter`. Just have a look at the revised adapter code, which has just one change.

```
01
    class paypalAdapter implements paymentAdapter {
02
03
        private $paypal;
04
05
        public function __construct(PayPal $paypal) {
06
            $this->paypal = $paypal;
07
        }
08
09
        public function pay($amount) {
10
            $this->paypal->payAmount($amount);
11
        }
    }
12
```

So just one change and we are there.

Adding a New Adapter

At this point, we've seen how we can use the adapter design patten to overcome the aforementioned scenarios. Now, it's very easy to add a new class dependent on the existing adapter. Let's say the MoneyBooker API is there for payment.

Then instead of using the MoneyBooker class directly, we should be applying the same adapter pattern we just used for PayPal.

```
01  // Concrete Implementation of MoneyBooker Class
02  class MoneyBooker {
03
04      public function __construct() {
05          // Your Code here //
06      }
07
08      public function doPayment($amount) {
09          // Paying via MoneyBooker //
10          echo "Paying via MoneyBooker: ".  $amount;
11      }
12  }
13
14  // MoneyBooker Adapter
15  class moneybookerAdapter implements paymentAdapter {
16
17      private $moneybooker;
```

```
17
18      public function __construct(MoneyBooker $moneybooker) {
19          $this->moneybooker = $moneybooker;
20      }
21
22      public function pay($amount) {
23          $this->moneybooker->doPayment($amount);
24      }
25  }
26
27  // Client Code
28  $moneybooker = new moneybookerAdapter(new MoneyBooker());
29  $moneybooker->pay('2629');
30
```

As you can see, the same principles apply. You define a method that's available to third-party classes and then, if a dependency changes its API, you simply change the dependent class without exposing its external interface.

Conclusion

A great application is constantly hooked into other libraries and APIs, so I would propose that we implement the adapter method, so that we do not experience any trouble when a third-party API or library changes its code base.