Behavior Parameterization

In previous post we saw how to cope up with requirement change and wrote a code that is worse. So now we will introduce a concept that will help us in coping with requirement change and will get rid of boilerplate code.

In case you are wondering what is boilerplate code – they are lines of code that does not add anything to business logic. Example: for loop. Who cares about for loop unless you are returning the proper values to client?

Now let us think of better solution of the problem i.e. to get apples of specified color and get apples of specified weight. Notice there is one thing in common in both of requirements i.e. decision is made i.e. boolean value is returned.

Predicate –

Predicate as we know it is a Boolean-valued function in mathematical logic.

Predicate P: X→ {true, false} is called predicate on X.

Read about [Predicate Functional Interface here](http://data-structure-learning.blogspot.com/2015/07/java-lambda-predicate-functional.html).

Let us write Predicate for Apple with a method test. So let us extract an interface with test() method in ApplePredicate Interface.

ApplePredicate Interface

**public** **interface** ApplePredicate {

**boolean** test(Apple apple);

}

Now let us implement this interface in **AppleWeightPredicate class**.

**public** **class** AppleWeightPredicate **implements** ApplePredicate {

@Override

**public** **boolean** test(Apple apple) {

**return** apple.getWeight() > 50;

}

}

Now let us implement this interface in **AppleColorPredicate class**.

**public** **class** AppleColorPredicate **implements** ApplePredicate {

@Override

**public** **boolean** test(Apple apple) {

**return** apple.getColor().equalsIgnoreCase("green");

}

}

Now we will write the filter method with List<Apple> and ApplePredicate as parameter. As ApplePredicate is implemented by AppleWeightPredicate and AppleColorPredicate we can pass any instance of those classes. This leads to different behavior. So we are actually passing a behavior as a parameter.

**public** **static** List<Apple> filter(**final** List<Apple> apples, **final** ApplePredicate predicate) {

List<Apple> result=**new** ArrayList<Apple>();

**for** (Apple apple : apples) {

**if**(predicate.test(apple)){

result.add(apple);

}

}

**return** result;

}

This method can be called as:

*filter*(apples, **new** AppleColorPredicate());

*filter*(apples, **new** AppleWeightPredicate());

Now let us combine AppleColorPredicate and AppleWeightPredicate class as AppleWeightAndColorPredicate class(apple weight greater than 50 and color must be green).

**public class AppleWeightAndColorPredicate implements ApplePredicate {**

**public boolean test(Apple apple) {**

**return** apple.getWeight() > 50

&& apple.getColor().equalsIgnoreCase("green");

**}**

**}**

Excellent we just came up with good solution. But there is still a problem of boilerplate code. Bold denotes the boilerplate code. This code does not add anything in our business logic but we have to write it.

Does anonymous inner class solve our problem? Let’s see.

List<Apple> result = *filter*(list, **new ApplePredicate() {**

**@Override**

**public boolean test(Apple apple) {**

**return** apple.getColor().equalsIgnoreCase("green");

**}**

**});**

So did the anonymous inner class solve your problem? I don’t think so. It just increased it by some weird syntax.

So what is the solution to this? First, I will throw a Java 8 lambda operator “**->**” here.

List<Apple> greenApples = *filter*(list, (Apple apple) -> apple.getColor().equals("green"));

What does this do? So this means that filter out apples with color green. We can use this Lambda Operator if the interface is Functional Interface. To read about Functional Interface click [here](http://data-structure-learning.blogspot.com/2015/06/functional-interfaces-java-8.html) and [here](http://data-structure-learning.blogspot.com/2015/06/functional-interface-without-arguments.html) and [here](http://data-structure-learning.blogspot.com/2015/06/functional-interface-with-arguments.html).

Now let us change our Predicate interface for generic type T.

**public** **interface** Predicate<T> {

**boolean** test(T t);

}

Let us change our filter method too for using generic type T.

**public** **static** <T> List<T> filter(**final** List<T> list,

**final** Predicate<T> predicate) {

List<T> result = **new** ArrayList<T>();

**for** (T element : list) {

**if** (predicate.test(element)) {

result.add(element);

}

}

**return** result;

}

List<Apple> greenApples = *filter*(list, (Apple apple) -> apple.getColor().equals("green"));

List<Apple> redApples = *filter*(list, (Apple apple) -> apple.getColor().equals("red"));

List<Apple> appleWeight = *filter*(list, (Apple apple) -> apple.getWeight() > 50);

List<Integer> evenNumbers = *filter*(ints, (i) -> i % 2 == 0);

See how we passed behavior in the method parameter rather than writing lot of code in a class and then passing instance of the class.

So this is Behavior Parameterization.