http://www.yegor256.com/2015/02/26/composable-decorators.html

Composable Decorators vs. Imperative Utility Methods

26 February 2015 modified on 5 May 2015 Yegor Bugayenko

The decorator pattern is my favorite among all other patterns I'm aware of. It is a very simple and yet very powerful mechanism to make your code highly cohesive and loosely coupled. However, I believe decorators are not used often enough. They should be everywhere, but they are not. The biggest advantage we get from decorators is that they make our code *composable*. That's why the title of this post is composable decorators. Unfortunately, instead of decorators, we often use imperative utility methods, which make our code procedural rather than object-oriented.

© Матрёшка

First, a practical example. Here is an interface for an object that is supposed to read a text somewhere and return it:

```
interface Text {
  String read();
}
```

Here is an implementation that reads the text from a file:

```
final class TextInFile implements Text {
  private final File file;
  public TextInFile(final File src) {
    this.file = src;
  }
  @Override
  public String read() {
    return new String(
      Files.readAllBytes(), "UTF-8"
    );
  }
}
```

And now the decorator, which is another implementation of `Text` that removes all unprintable characters from the text:

```
final class PrintableText implements Text {
  private final Text origin;
  public PrintableText(final Text text) {
    this.origin = text;
  }
  @Override
  public String read() {
    return this.origin.read()
      .replaceAll("[^\p{Print}]", "");
  }
}
```

Here is how I'm using it:

```
final Text text = new PrintableText(
  new TextInFile(new File("/tmp/a.txt"))
);
String content = text.read();
```

As you can see, the `PrintableText` doesn't read the text from the file. It doesn't really care where the text is coming from. It *delegates* text reading to the encapsulated instance of `Text`. How this encapsulated object will deal with the text and where it will get it doesn't concern `PrintableText`.

Let's continue and try to create an implemetation of `Text` that will capitalize all letters in the text:

```
final class AllCapsText implements Text {
  private final Text origin;
  public AllCapsText(final Text text) {
    this.origin = text;
  }
  @Override
  public String read() {
    return this.origin.read().toUpperCase(Locale.ENGLISH);
  }
}
```

How about a `Text` that trims the input:

```
final class TrimmedText implements Text {
  private final Text origin;
  public TrimmedText(final Text text) {
    this.origin = text;
  }
  @Override
  public String read() {
    return this.origin.read().trim();
  }
}
```

I can go on and on with these decorators. I can create many of them, suitable for their own individual use cases. But let's see how they all can play together. Let's say I want to read the text from the file, capitalize it, trim it, and remove all unprintable characters. And I want to be *declarative*. Here is what I do:

```
final Text text = new AllCapsText(
  new TrimmedText(
    new PrintableText(
      new TextInFile(new File("/tmp/a.txt"))
    )
  )
);
String content = text.read();
```

First, I create an instance of `Text`, *composing* multiple decorators into a single object. I declaratively define the behavior of `text` without actually executing anything. Until method `read()` is called, the file is not touched and the processing of the text is not started. The object `text` is just a composition of decorators, not an executable *procedure*. Check out this article about declarative and imperative styles of programming: Utility Classes Have Nothing to Do With Functional Programming.

This design is much more flexible and reusable than a more traditional one, where the `Text` object is smart enough to perform all said operations. For example, class `String` from Java is a good example of a bad design. It has more than 20 *utility methods* that should have been provided as decorators instead: `trim()`, `toUpperCase()`, `substring()`, `split()`, and many others, for example. When I want to trim my string, uppercase it, and then split it into pieces, here is what my code will look like:

```
final String txt = "hello, world!";
final String[] parts = txt.trim().toUpperCase().split(" ");
```

This is imperative and procedural programming. Composable decorators, on the other hand, would make this code object-oriented and declarative. Something like this would be great to have in Java instead (pseudo-code):

```
final String[] parts = new String.Split(
  new String.UpperCased(
    new String.Trimmed("hello, world!")
  )
);
```

To conclude, I recommend you think twice every time you add a new utility method to the interface/class. Try to avoid utility methods as much as possible, and use decorators instead. An ideal interface should contain only methods that you absolutely cannot remove. Everything else should be done through composable decorators.