Poznan University of Technology

Object Oriented Programming

Jakub Piotr Hamerliński, M.Eng.

Object Oriented Programming

Agenda

- Decorator pattern
- Task

Decorator pattern

Object Oriented Programming

Decorator pattern

The decorator pattern is a fairly straightforward yet effective tool for creating highly coherent, loosely connected code. But in my opinion, decorators are not utilised enough. They should be all over, yet they are not. The fact that decorators make our code composable is the main benefit we receive from them. Unfortunately, we frequently substitute imperative utility methods for decorators, which results in procedural rather than object-oriented code.

Object Oriented Programming

Decorator pattern

Here is an interface for an object that is designed to read and return text from somewhere:

```kotlin
//kotlin
interface Text {
  fun readText(): String
}
```

```kotlin
//kotlin
class TextInFile(private val file: File) : Text {
  override fun readText(): String {
    return String(Files.readAllBytes(file.toPath()))
  }
}
```

Object Oriented Programming
Decorator pattern

Now using decorator, that is another Text implementation, we eliminate all unprintable characters from the text:

```kotlin
//kotlin
class PrintableText(private val text: Text) : Text {
  override fun readText(): String {
    return this.text.readText().replace(Regex("[^\\p{Print}]"), "")
  }
}
```

Here's how we'll put it to use:

```kotlin
//kotlin
val text = PrintableText(TextInFile(File("src/poem.txt")))
print(text.readText())
```

Object Oriented Programming
Decorator pattern

Let's keep going and try to make a Text implementation that capitalizes all letters in the text:

```kotlin
//kotlin
class AllCapsText(private val text: Text) : Text {
  override fun readText(): String {
    return this.text.readText().uppercase()
  }
}
```

Here's how we'll put it to use:

```kotlin
//kotlin
val text: Text = AllCapsText(
  PrintableText(
    TextInFile(File("src/tmp/a.txt"))
  )
)
val content: String = text.readText()
```

Object Oriented Programming

Decorator pattern

This design is far more adaptable and reusable than a more traditional one in which the Text object is efficient enough to perform all of the aforementioned operations. String from Java, for instance, is a good example of a poor design. It contains over 20 utility methods that should have been provided as decorators instead, such as trim(), toUpperCase(), substring(), split(), and many others.

Object Oriented Programming

Decorator pattern

tldr;

When you need to be able to provide extra behaviors to objects at runtime without disrupting the code that utilizes these objects, use the Decorator pattern. The Decorator allows you to layer your business logic, define a decorator for each layer, and construct objects at runtime with various combinations of this logic. Because they all have the same interface, the client programs may treat them all the same.

Task

Object Oriented Programming

Task

Task Here's how my code (bad way! boo!) will look when I want to trim, uppercase, and split a string into pieces:

```
1  //java
2  final String txt = "HELLO, WORLD!  ";
3  final String[] parts = txt.trim().toLowerCase().split(" ");
```

This is procedural and imperative programming. Composable decorators, on the other hand, would turn this code into something more object-oriented and declarative:

```
1  //kotlin
2  val parts: Array<CharSeq> = Split(
3    LowerCased(
4      Trimmed("HELLO, WORLD!  ")
5    )
6  )
```

Object Oriented Programming

Task

Using examples and gathered knowledge, create:

- an interface *Sequence* with method *printedSeq* returning string,
- a class *Characters* implementing *Sequence*,
- the following decorators of *Characters*:
    - *LowerCased*
    - *Substring*
    - *Concatenation*

Sources

Object Oriented Programming

Based on:

- https://www.yegor256.com/2015/02/26/composable-decorators.html
- https://www.yegor256.com/2017/01/31/decorating-envelopes.html
- https://www.yegor256.com/2015/10/01/vertical-horizontal-decorating.html
- https://www.yegor256.com/2017/10/10/streams-vs-decorators.html
- https://refactoring.guru/design-patterns/decorator