Poznan University of Technology
Object Oriented Programming
Decorator pattern

Jakub Piotr Hamerliński, M.Eng.

https://www.linkedin.com/in/hamerlinski

https://github.com/hamerlinski

Object Oriented Programming

Agenda

01. Decorator pattern
02. Task

> "Who says exactly what they're thinking? What kind of game is that?"

Kelly Kapoor

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:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <memory>
#include <algorithm>
#include <cctype>
#include <string>
class Text {
 public:
  virtual ~Text() = default;
  virtual std::string Content() = 0;
};
```

Object Oriented Programming

Decorator pattern

Next, implementation of the Text interface:

```
class TextInFile : public Text {
 public:
  explicit TextInFile(const std::string& file_path) : file_path_(file_path) {}
  std::string Content() override {
    std::ifstream file(file_path_);
    std::stringstream buffer;
    buffer << file.rdbuf();
    return buffer.str();
  }
 private:
  std::string file_path_;
};
```

Object Oriented Programming

Decorator pattern

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

```
class PrintableText : public Text {
 public:
  explicit PrintableText(std::unique_ptr<Text> text) : text_(std::move(text)) {}
  std::string Content() override {
    std::string content = text_->Content();
    content.erase(std::remove_if(content.begin(), content.end(), [](char c) { return !std::isprint(c); }), content
    return content;
  }
 private:
  std::unique_ptr<Text> text_;
};
```

Object Oriented Programming

Decorator pattern

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

```
class AllCapsText : public Text {
 public:
  explicit AllCapsText(std::unique_ptr<Text> text) : text_(std::move(text)) {}
  std::string Content() override {
    std::string content = text_->Content();
    std::transform(content.begin(), content.end(), content.begin(), ::toupper);
    return content;
  }
 private:
  std::unique_ptr<Text> text_;
};
```

Object Oriented Programming

Decorator pattern

Let's continue and trim characters from text:

```
class TrimmedText : public Text {
 public:
  explicit TrimmedText(std::unique_ptr<Text> text) : text_(std::move(text)) {}
  std::string Content() override {
    std::string content = text_->Content();
    content.erase(content.begin(), std::find_if(content.begin(), content.end(), [](unsigned char c) { return !std:
    content.erase(std::find_if(content.rbegin(), content.rend(), [](unsigned char c) { return !std::isspace(c); }).
    return content;
  }
 private:
  std::unique_ptr<Text> text_;
};
```

Object Oriented Programming

Decorator pattern

We finish by adding example usage:

```
int main() {
  // Create a decorated Text object
  std::unique_ptr<Text> text = std::make_unique<AllCapsText>(
      std::make_unique<TrimmedText>(
          std::make_unique<PrintableText>(
              std::make_unique<TextInFile>("sample.txt"))));
  // Content the text using the decorated Text object
  std::string content = text->Content();
  // Output the content
  std::cout << content << std::endl;
  return 0;
}
```

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

Create a program that processes a sequence of integers using the decorator pattern. The program should have the following components:

An `IntegerSequence` interface with a `Numbers()` method that returns a `std::vector<int>`. A `VectorSequence` concrete component that implements the `IntegerSequence` interface, reading integers from a `std::vector<int>` .

Object Oriented Programming

Task

Three decorators:
`PositiveSequence`: Removes negative numbers from the sequence.
`EvenSequence`: Removes odd numbers from the sequence.
`SortedSequence`: Sorts the sequence in ascending order.

Compose these decorators together to create an IntegerSequence object with combined functionality.

Object Oriented Programming

Task

Hints:

01. Start by defining the IntegerSequence interface with a virtual Numbers() method.
02. Implement the VectorSequence concrete component that reads integers from a `std::vector<int>`.
03. Implement the three decorators PositiveSequence, EvenSequence, and SortedSequence that inherit from IntegerSequence. Each decorator should override the Numbers() method to modify the sequence according to its specific behavior.
04. In the main() function, create a VectorSequence object with a given sequence of integers, and compose the decorators together to process the sequence.

Thank you

Feel free to reach me via LinkedIn

Fin