Design Patterns - Lab 3

Composite

Goal

Introduce the basic concept of the Composite pattern and apply Composite pattern to our book.

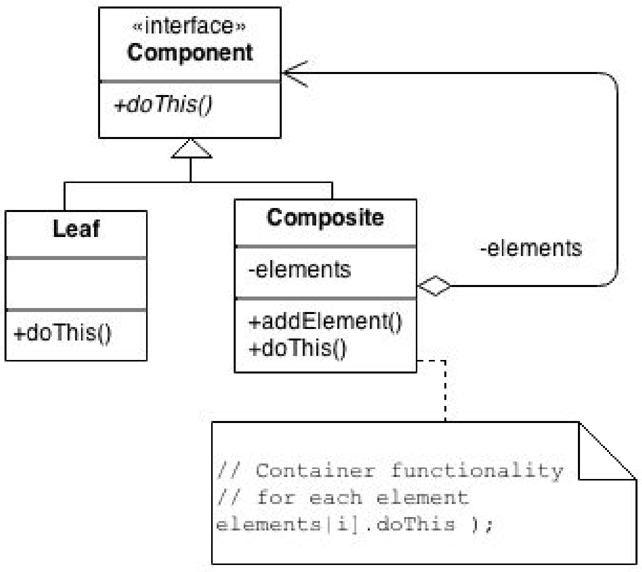
Tasks:

1. We want to enhance the design and fix the other design flaws identified in the previous lab

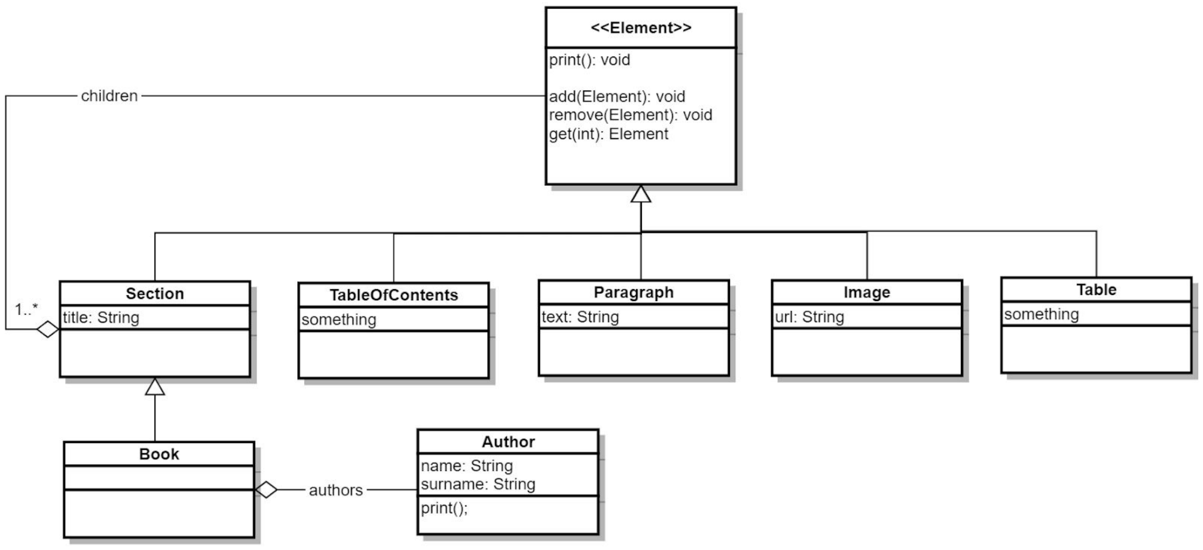
2. First thing, we would like to have the ability to add one or more subchapters to subchapters and so on (open ended number of subchapters), and would like to handle chapters and subchapters classes uniformly in our project. How can we do this?

3. 💡 Idea: think of the structure of the directories and files. A directory can contain multiple directories and files. Do you know any design pattern that can help?

4. What about Composite design pattern? We should treat individual objects and compositions of objects a.k.a "composites" in the same way. The diagram of the Composite pattern is:



5. Change the structure of the project in order to match the next diagram:



6. Implement this new diagram. Some points that you need to consider:
    a. What implementation should be given to composite-related methods (add, remove, get) in leaf classes?
    b. What data structure do you need to hold the children? (List probably as in this case children need to be ordered)
    c. What does Section.print do? How does iterate through all children?
    d. Be careful how you override Book.print (don't forget to call super.print)

7. Add the main function to your implementation and add the missing implementation in order to compile and execute the main function code

```
public static void main(String[] args) {
  Book noapteBuna = new Book("Noapte buna, copii!");
  Author rpGheo = new Author("Radu Pavel Gheo");
  noapteBuna.addAuthor(rpGheo);

  Section cap1 = new Section("Capitolul 1");
  Section cap11 = new Section("Capitolul 1.1");
  Section cap111 = new Section("Capitolul 1.1.1");
  Section cap1111 = new Section("Subchapter 1.1.1.1");
  noapteBuna.addContent(new Paragraph("Multumesc celor care ..."));
  noapteBuna.addContent(cap1);
  cap1.add(new Paragraph("Moto capitol"));
  cap1.add(cap11);
  cap11.add(new Paragraph("Text from subchapter 1.1"));
  cap11.add(cap111);
  cap111.add(new Paragraph("Text from subchapter 1.1.1"));
  cap111.add(cap1111);
  cap1111.add(new Image("Image subchapter 1.1.1.1"));

  noapteBuna.print();
}
```

8. The expected output is:

```
Book: Noapte buna, copii!

Authors:
Author: Radu Pavel Gheo

Paragraph: Multumesc celor care ...
Capitolul 1
Paragraph: Moto capitol
Capitolul 1.1
Paragraph: Text from subchapter 1.1
Capitolul 1.1.1
Paragraph: Text from subchapter 1.1.1
Subchapter 1.1.1.1
Image with name:Image subchapter 1.1.1.1
```

9. Let's analyse our second design.

> *Remember: Design (of any kind) is an iterative process and critical feedback is always a powerful tool.*

What are the flaws / "smells" / issues / problems with this design.List as many you find below:

a. we transformed the composition relationships between Book <> Chapter, Chapter <> Subchapter etc. into aggregation between Section <> Element => (i) it is possible to add an element (section, paragraph etc) to multiple sections, or even books, (ii) don't get destroyed alongside its parent
b. …………………………………………………………………………………..
c. …………………………………………………………………………………..

10. How can we fix the composition vs aggregation issue?

    💡 Idea: think of the structure of the directories and files. Is it possible to have a file in multiple directories?

11. Solution 1: Clone each element when added. The Section.add method will create a clone of its Element argument, hence storing a copy of the original element, each section handling its own elements. The drawback with this method is that Elements need to be cloneable, which is not always easy/possible.

12. Solution 2: Disallow sharing elements. The Section.add method will check whether its Element argument is already added to another section and throw an appropriate exception in this case. Each Element needs to keep a reference to its parent (this is both good as it will allow us to traverse the data structure bottom-up too, but it will add an additional memory overhead to each Element instance and Element will be transformed from an interface to an abstract class). The drawback of this approach is that it won’t fix all the issues introduced from transforming a composition into an aggregation, such as the lifecycle of the child elements.

13. Pick a solution that you like better and implement it!

14. Don’t forget to commit and push your code to your github account.