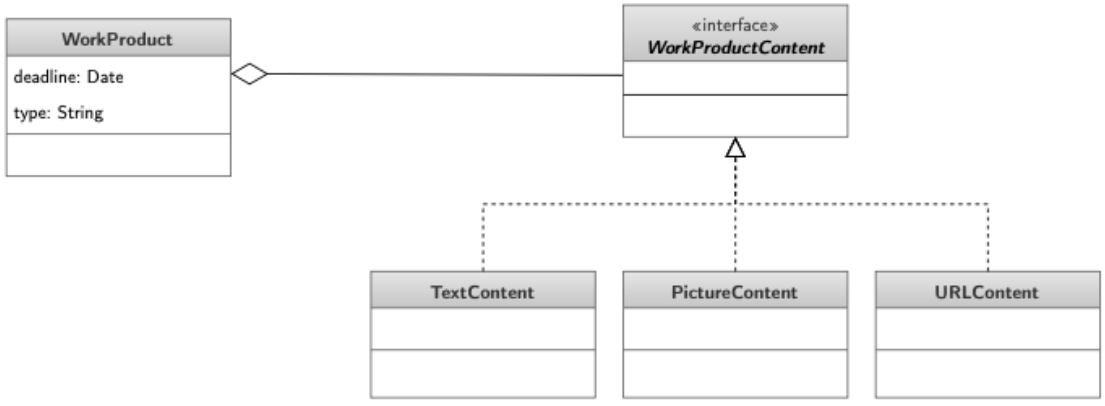
Lab 6: Design Patterns and Principles

1. Consider a workflow system supporting software developers. The system enables managers to model the process the developers should follow in terms of processes and work products. The manager can assign specific processes to each developer and set deadlines for the delivery of each work product. The system supports several types of work products (which is expected to expand in the future), including formatted text, picture, and URLs. The manager, while editing the workflow, can dynamically set the type of each work product at run time.

   What design pattern would you use to facilitate this design? Justify your answer. Use a UML class diagram (with the necessary attributes and operations) to describe your suggested solution. Use code excerpts where necessary.

   For this situation, it is appropriate to use the Bridge design pattern. The interface class stores the deadline and type of work product and provides the interface to the rest of the application. The implementor classes store the content of the work product and can be substituted at run time. All implemented classes comply with the same abstract WorkProductContent interface.

   A sample application of the Bridge pattern for this scenario is shown below as a UML class diagram.

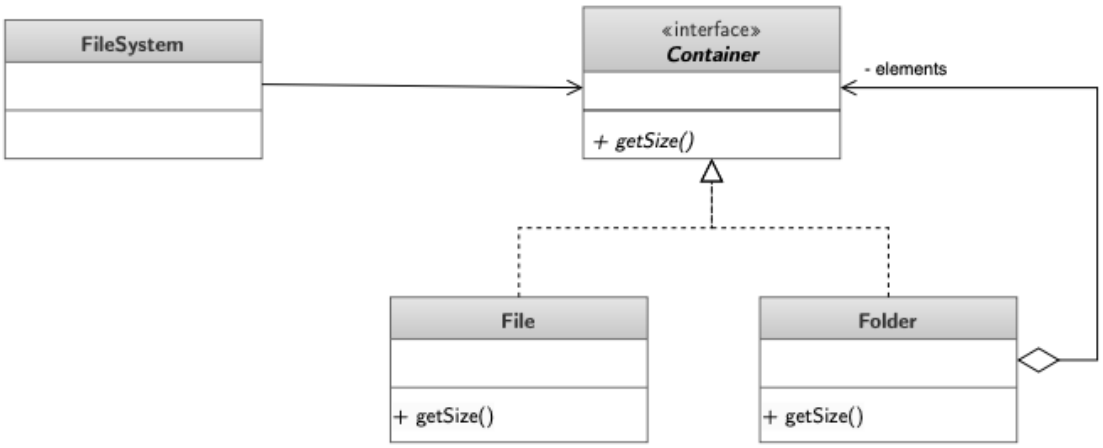


2. Suppose that you are designing a software application that needs to provide a mechanism (operation) to return the size of a folder (directory). Directories may contain files or other directories. Files such as shortcuts should not be counted in the computation of size though.

   What design pattern would you use to facilitate this design? Justify your answer. Use a UML class diagram (with the necessary attributes and operations) to describe your suggested solution. Use code excerpts where necessary.

For this situation, it is appropriate to use the Composite design pattern because we are going to compose objects into tree structures to represent part-whole hierarchies. We also need to treat individual objects and compositions of objects uniformly.

A sample application of the Composite pattern for this scenario is shown below as a UML class diagram.



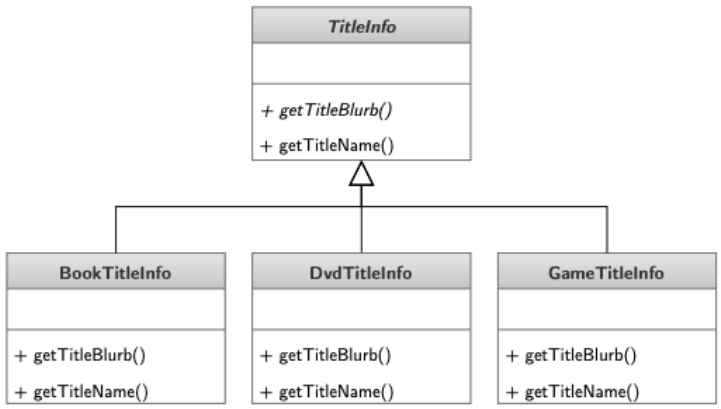
3. Which design pattern is used in the following Java code fragment? When identifying the design pattern that is used, precisely identify the classes in this code fragment that play the different roles in the pattern you recognize. Use a UML class diagram (with the necessary attributes and operations) showing the relations between these classes and their (design pattern) roles.

```
1  public abstract class TitleInfo {
2    private String titleName;
3
4    public final String ProcessTitleInfo () {
5      StringBuffer titleInfo = new StringBuffer();
6       titleInfo .append(this. getTitleBlurb ());
7       titleInfo .append(this.getDvdEncodingRegion Info());
8      return titleInfo . toString ();
9    }
10
11   public final void setTitleName(String titleNameIn) {
12     this .titleName = titleNameIn;
13   }
14   public final String getTitleName() {
15     return this .titleName;
16   }
17
18   public abstract String getTitleBlurb ();
19
20   public String getDvdEncodingRegionInfo() {
21     return " ";
22   }
23 }
24
```

```
25 public class DvdTitleInfo extends TitleInfo {
26   private String star;
27   private char encodingRegion;
28
29   public DvdTitleInfo(String titleName,
30     String star,
31     char encodingRegion) {
32     this.setTitleName(titleName);
33     this.setStar(star);
34     this.setEncodingRegion(encodingRegion);
35   }
36   public void setStar(String starIn) {
37     this.star = starIn;
38   }
39   public String getStar() {
40     return this.star;
41   }
42   public void setEncodingRegion(char encodingRegionIn) {
43     this.encodingRegion = encodingRegionIn;
44   }
45   public char getEncodingRegion() {
46     return this.encodingRegion;
47   }
48   public String getTitleBlurb() {
49     return ("DVD: " + this.getTitleName() +
50       ", starring " + this.getStar());
51   }
52   public String getDvdEncodingRegionInfo() {
53     return (", encoding region: " +
54       this.getEncodingRegion());
55   }
56 }
57
58 public class BookTitleInfo extends TitleInfo {
59   private String author;
60
61   public BookTitleInfo(String titleName,
62     String author) {
63     this.setTitleName(titleName);
64     this.setAuthor(author);
65   }
66   public void setAuthor(String authorIn) {
67     this.author = authorIn;
68   }
69   public String getAuthor() {
70     return this.author;
71   }
72   public String getTitleBlurb() {
73     return ("Book: " + this.getTitleName() +
74       ", Author: " + this.getAuthor());
75   }
76 }
78 public class GameTitleInfo extends TitleInfo {
79   public GameTitleInfo(String titleName) {
80     this.setTitleName(titleName);
81   }
82
83   public String getTitleBlurb() {
84     return ("Game: " + this.getTitleName());
85   }
86 }
```

The Template Method pattern used in this code fragment. getTitleBlurb() is the template method.

A sample application of the Template Method pattern for this scenario is shown below as a UML class diagram.

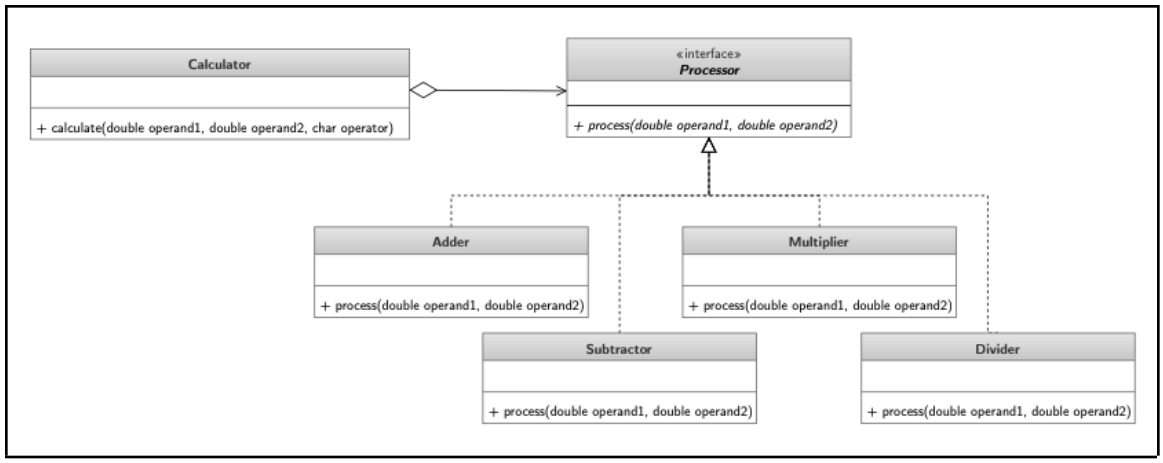


4. Consider the implementation of a basic calculator from **Lab #3** shown below. Recall that this code was developed by a new graduate and recalls making similar, simple calculator programs as part of their studies, so they created a simple program where the main method of the program takes two operands separated by an operator. These are used by the calculate method to perform the operation related to the operator provided. Recall also that we calculated the cyclomatic complexity of calculate method to be 6.

   By adopting a design pattern, we can reduce the cyclomatic complexity of calculate method. The Strategy pattern allows the definition of a family of algorithms such that they are interchangeable within an application. Thus, it lets algorithms vary independently from the clients that use it. Strategy is typically used to configure a class with one of many behaviors, when different variants of an algorithm are needed or when a class defines many behaviors and these appear as multiple conditional statements in its operations (something we can see in the initial calculate method).

   Produce a refactored design by applying the Strategy pattern to the calculate method for the basic calculator code shown below. Use a UML class diagram (with the necessary attributes and operations) to depict the refactored design

A sample application of the Strategy pattern to refactor the design of the calculate method is shown below as a UML class diagram.


Calculator
+ calculate(double operand1, double operand2, char operator)
«interface»
Processor
+ process(double operand1, double operand2)
Adder
+ process(double operand1, double operand2)
Multiplier
+ process(double operand1, double operand2)
Subtractor
+ process(double operand1, double operand2)
Divider
+ process(double operand1, double operand2)