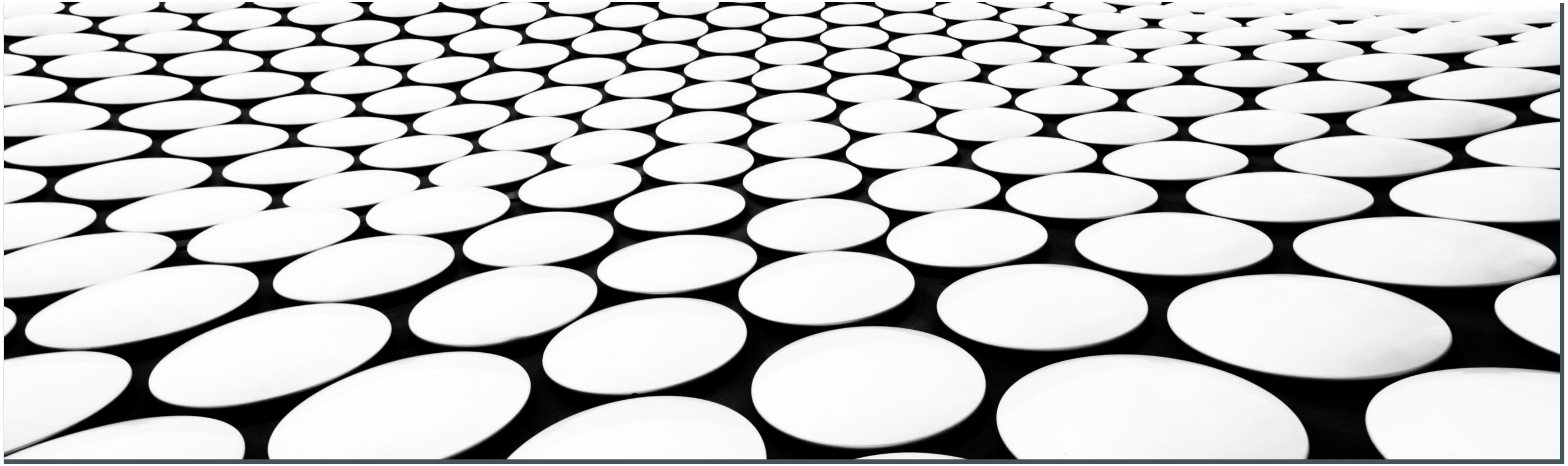
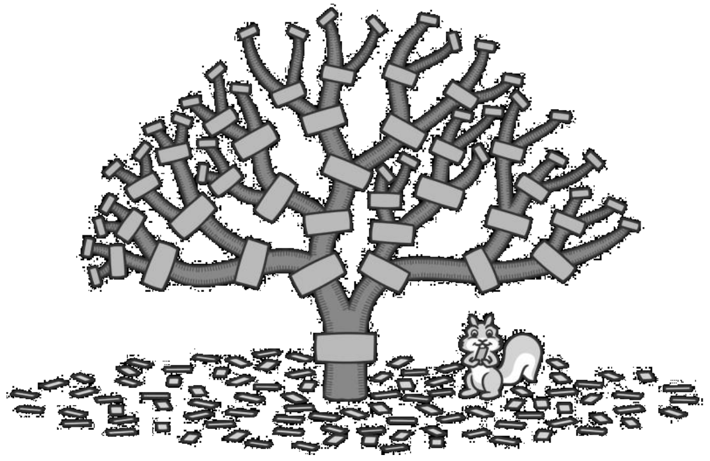
STRUCTURAL DESIGN PATTERNS

COMPOSITION, DECORATOR



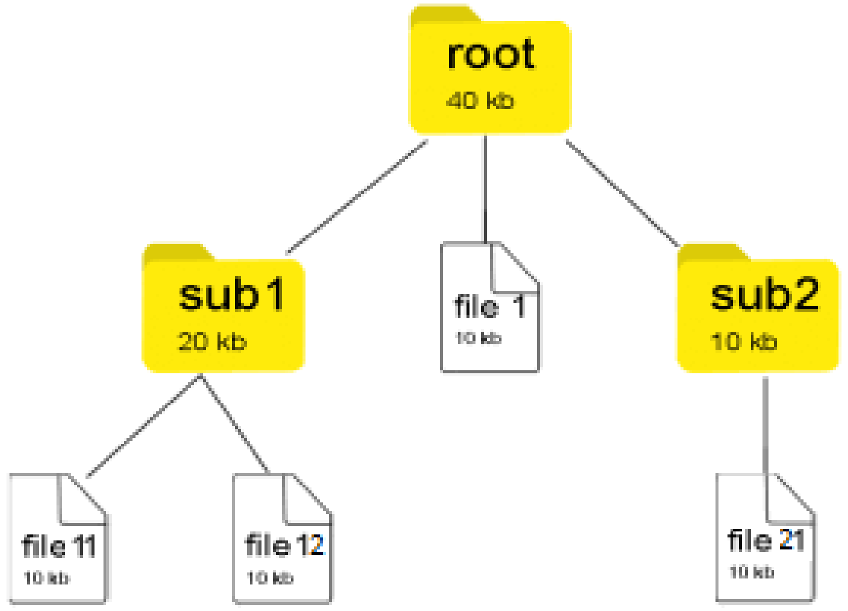
COMPOSITE PATTERN



- *Also known as: **Object Tree***
- Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.
- From the name, “Composite” means the combination or made from different parts. So this pattern provides solution to operate **group of objects and single object in similar way**.

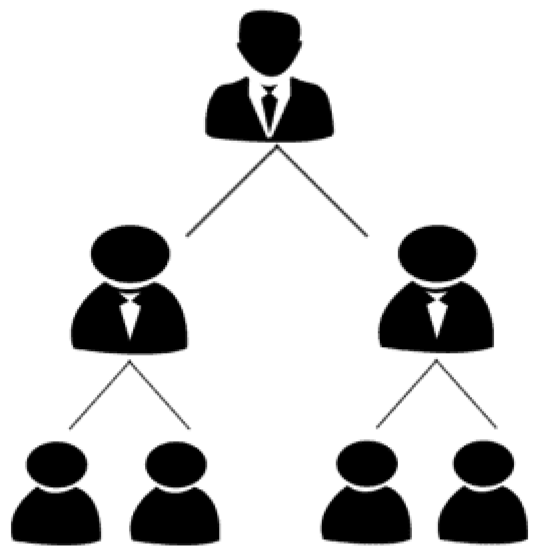
REAL LIFE EXAMPLES

- The most common example can be file-directory structure. As shown in image.
- Node can be a directory or a file.
- Even though they are different type of objects, many times we need to treat them similarly . For eg: we can check size of file as well as size of directory. The size of directory is sum of size of all the files in that directory.



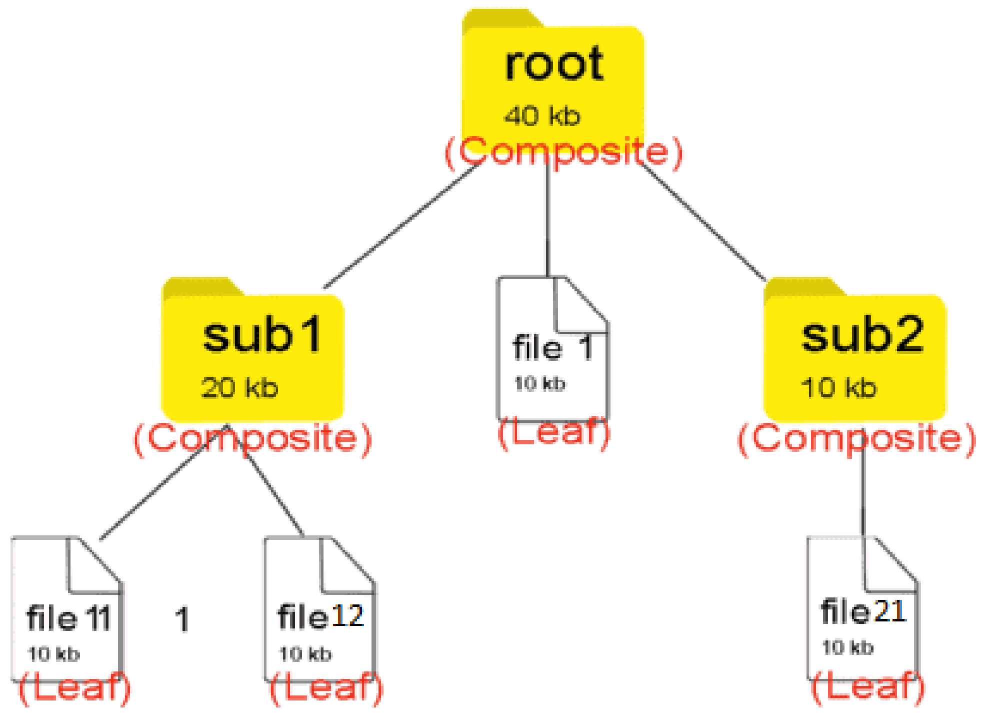
REAL LIFE EXAMPLES CONT…..

- Another example can be the employee hierarchy in an organization.
- In the image lets say we have CEO as top node, then mangers and then employees.
- CEO manages the mangers. And has group of employees.
- All employees, managers and the CEO are instances of a person.



COMPOSITE DESIGN PATTERN STRUCTURE

- The image shows the basic structure of the composite design pattern.
- There are 2 types of objects Composite object and Leaf object.
- 1) Composite Object : It is an object which contains other object. This has children e.g: Directory(file folder)
  2) **Leaf Object :** It is a single object. It does not have children e.g: File



WHERE CAN WE USE COMPOSITE PATTERN?

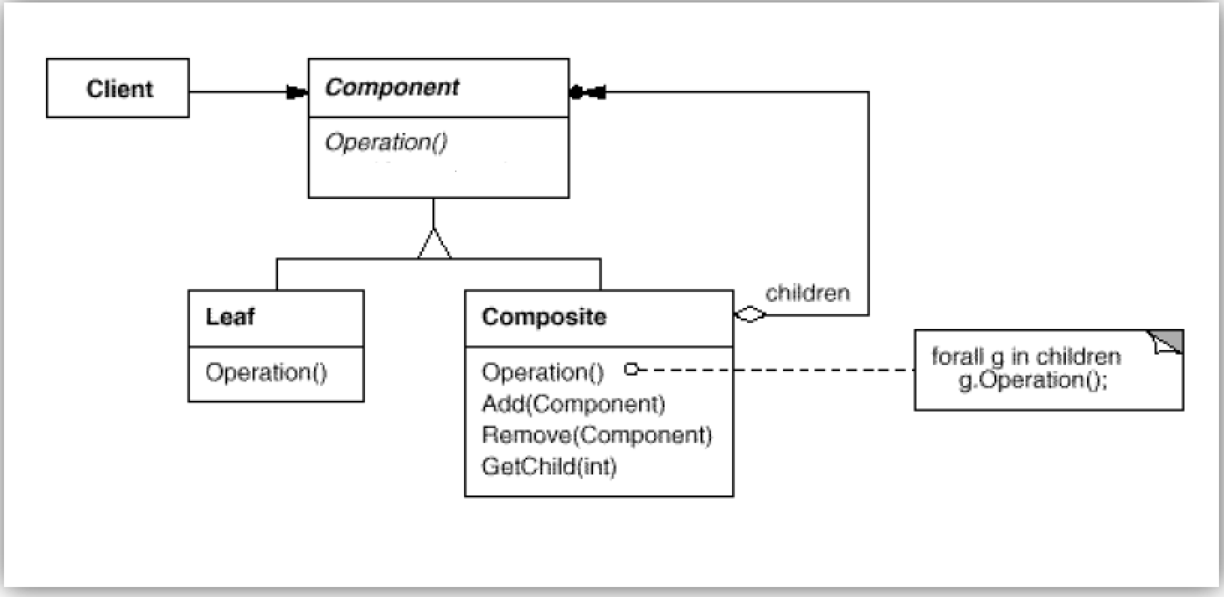
- When you want to represent a part-whole relationship in a tree structure (Objects can be represented in a tree structure hierarchy).
- Composite and individual objects are treated uniformly
- When we need parent-Child relationship

STEPS TO IMPLEMENT COMPOSITE DESIGN PATTERN

- We need to define leaf and composite objects as same data type
  - By implementing interface or extending class
- Then write(implement) common method in all the leaf and composite objects
- In Leaf node perform their own desired behavior
- In Composite object write customized function or each child of this node can call this function

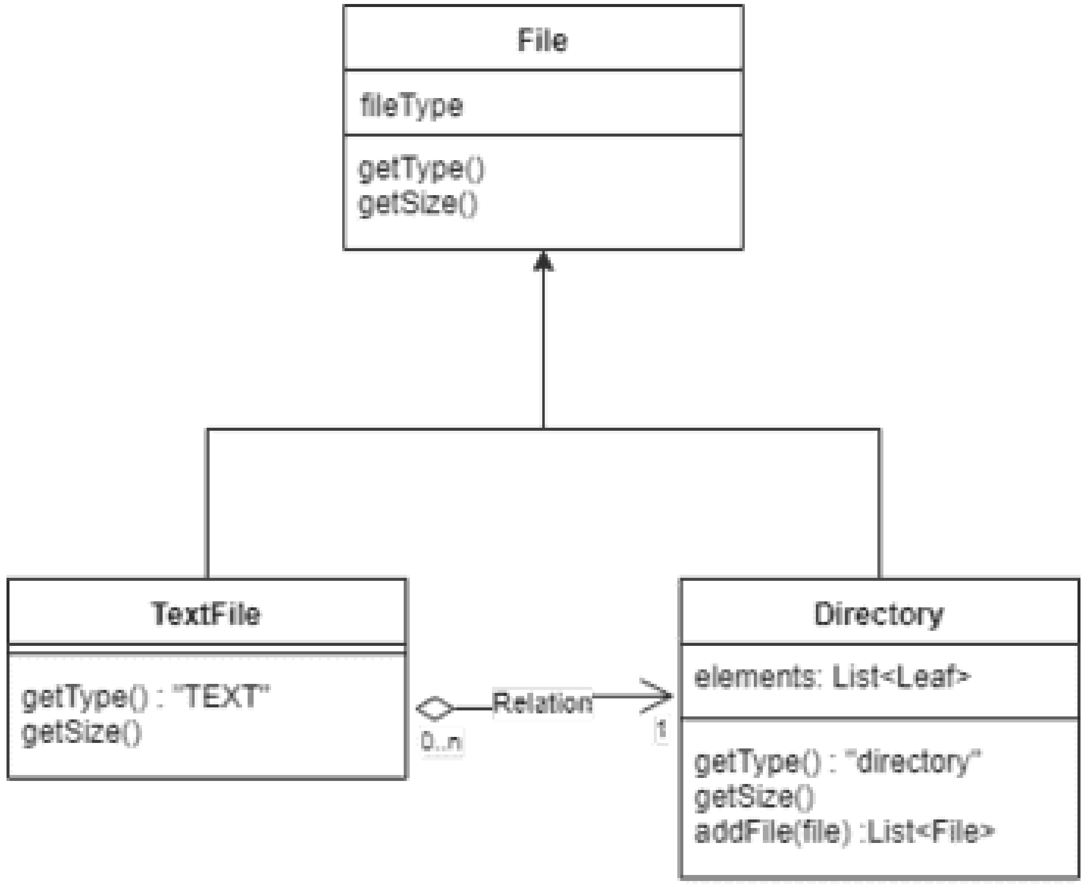
STRUCTURE & PARTICIPANTS

- Component
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all.
- Leaf
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- Composite
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client
  - manipulates objects in the composition through the Component interface.



UML DIAGRAM FOR COMPOSITE DESIGN PATTERN IMPLEMENTATION

- As shown in the image we will create interface for “File”. Then we will have two different classes. One for Directory type and second for other than directory like .txt, .doc, etc
  The Directory class is composite class that can contain other files or directories.



IMPLEMENTING COMPOSITE DESIGN PATTERN IN JAVA

- Let’s implement the example for file directory structure. We will create child1.txt and child2.txt files . Then we will have root directories. Root directories will contain child1.txt and child2.txt files.
- Then we will try to get size of child1.txt and root directory.
- As explained in step 1 lets create parent type as File interface

```
interface File {
    public String getType();
    public Long getSize();
}
```

- Now lets create classes for file and directory type

```
class TextFile implements File {
private Long size;

public TextFile(Long size) {
this.size = size;   }

public String getType() {
return “txt”;  }

public Long getSize() {
return this.size;  }
}
```

```
class Directory implements File {
private List <File> files = new ArrayList<>();

public String getType() {
return “directory”;   }

public void addFile(File file) {
files.add(file);   }

public Long getSize() {
Long size = 0L;

for (File file : files) {
size = size + file.getSize(); }

return size;   }
}
```

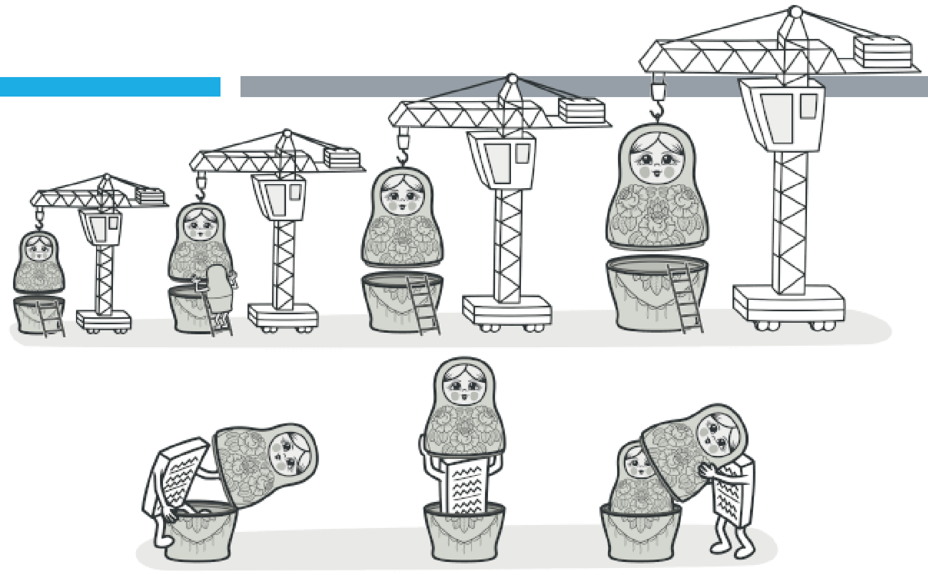
- Now we have implemented two leaf and one composite object. Both objects are of same time 'File'.
- We can use getSize() on TextFile class or Direcotry class.
- Now let's test the code using main method.

```
class Application {

        public static void main(String[] args) {
                TextFile child1 = new TextFile(100L);
                TextFile child2 = new TextFile(200L);

                Directory root= new Directory();
                root.addFile(child1);
                root.addFile(child2);

                System.out.println(child1.getSize());     //
output : 100
                System.out.println(root.getSize());    //
output : 300
        }
}
```

PROS AND CONS

- Pros
  - Data can be represented as tree structure
  - Same operations can be performed on different type of objects
  - Reduces the overhead of handling different type of objects
  - *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.
- Cons
  - Objects should be compatible
  - If the objects behavior is different we can not use it
  - Complexity can be increased

DECORATOR PATTERN



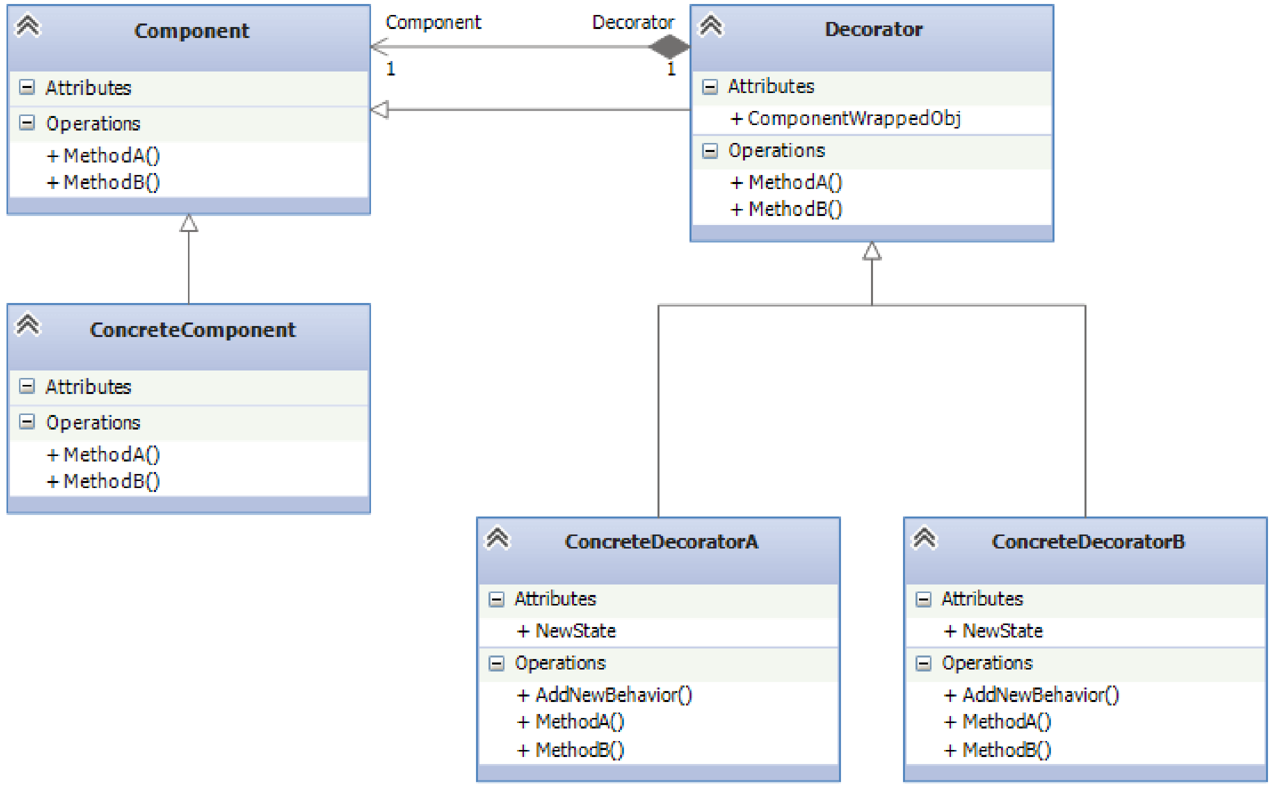
- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
- The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.
- The Decorator Pattern is also known as Wrapper.
- You can wrap a component with any number of decorators. (Wrapping a gift, putting it in a box, and wrapping the box.)
- Change the behavior of its component by adding new functionality before and/or after method calls to the component.
- Provides an alternative to subclassing for extending behavior.

WHERE TO USE

- Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
- Use the pattern when it’s awkward or not possible to extend an object’s behavior using inheritance.
  - Many programming languages have the final keyword that can be used to prevent further extension of a class. For a final class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

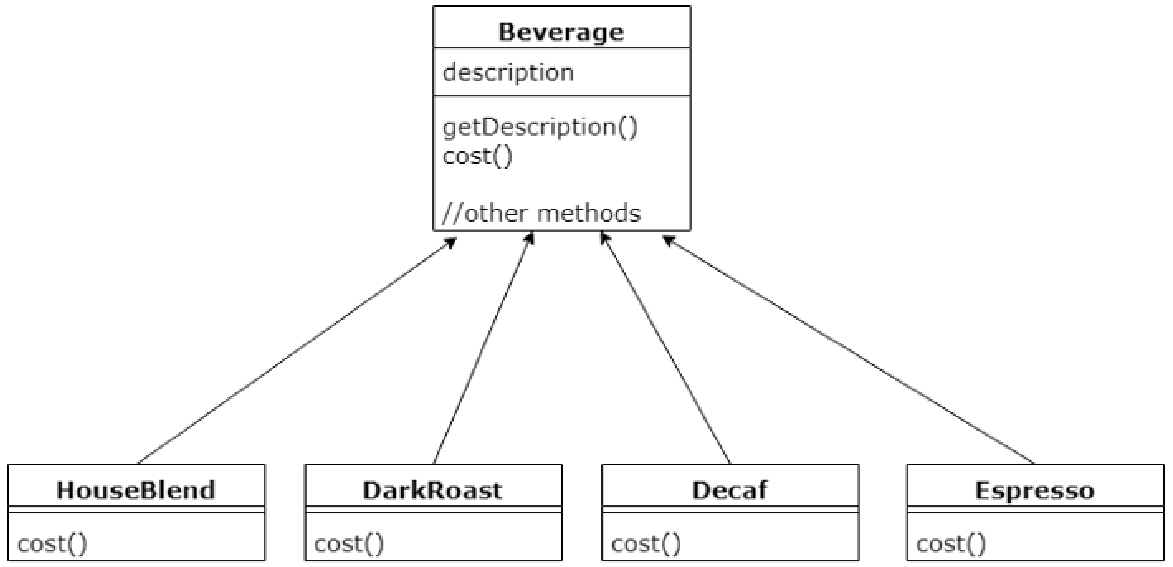
UML CLASS DIAGRAM & PARTICIPANTS

- The classes and objects participating in this pattern are:
- **Component** – Defines the interface for objects that can have responsibilities added to them dynamically.
- **Decorator** – The decorators implement the same interface(abstract class) as the component they are going to decorate. The decorator has a HAS-A relationship with the object that is extending, which means that the former has an instance variable that holds a reference to the later.
- **ConcreteComponent** – Is the object that is going to be enhanced dynamically. It inherits the Component.
- **ConcreteDecorator** – Decorators can enhance the state of the component. They can add new methods. The new behavior is typically added before or after an existing method in the component.



COFFEE SHOP EXAMPLE

- Suppose that you have a coffee shop, a. It serves four types of beverages: house blend, dark roast, decaf, and espresso.
- When you first started the business, you designed your classes like this:
- Beverage is abstract class and cost method is also abstract in it. Subclasses will implement it. Beverage has four sub-classes: HouseBlend, DarkRoast, Decaf, and Espresso. All of them override cost method and return their own cost. Here, we have used inheritance.
- In addition to coffee, you can also ask for condiments like steamed milk, soy, mocha, and whipped milk. You can implement it similar to before and use inheritance. This way we will have a huge number of classes: HouseBlendMilk, HouseBlendMocha, HouseBlendSoy, HouseBlendWhip, HouseBlendMilkMocha, HouseBlendMilkSoy, etc. The list goes on. There will be around 100 classes that we will have to make.
- That's what happen when we overdo inheritance. It can easily result in class explosion. There is clearly a better way to implement like this. Let's find one.



APPROACH 1

- In this approach we will add some boolean variables in Beverage class as flags. If it’s true, it means that condiment is present, so add its cost. This time cost() method is not abstract.
- In our concrete Beverage class like DarkRoast, we will override the cost method and add cost after calculating condiment cost by calling super.cost().
- Let’s say for Dark roast coffee, we will add 20 extra rupees.

```
@Override
public int cost(){
  return super.cost() + 20;
}
```

```
boolean milk;
boolean soy;
boolean mocha;
boolean whip;

// and their getters and setters

public int cost() {
  int total = 0;
  if (isMilk()) {
    total += MILK_COST;   }
  if (isSoy()) {
    total += SOY_COST;   }
  if (isMocha()) {
    total += MOCHA_COST;   }
  if (isWhip()) {
    total += WHIP_COST;   }
  return total; }
```

APPROACH 1

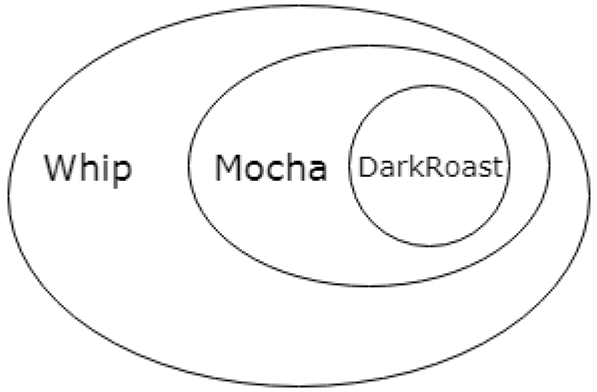
- Here’s how we will make HouseBlend milk coffee. Its output will be cost of house blend coffee + cost of steamed milk (condiment).

```
Beverage b = new HouseBlend();
b.setMilk(true);
System.out.println("total cost = "+b.cost());
```

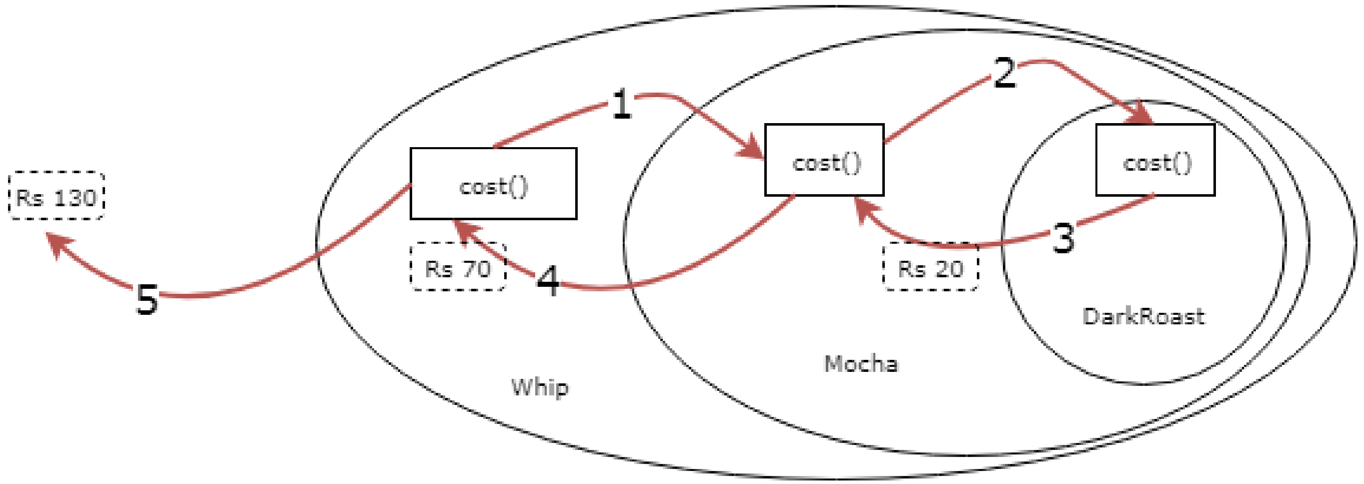
- Our code will work perfectly, but there are few problems with this simple approach:
- If there is a price change for condiment, or a new condiment is to be added, we will have to modify our existing code and make new methods. Note that adding new things should not alter existing code, as it may result in bugs.
- We may have some new beverages, like iced tea. For this, there is no point of adding milk condiment, but we will inherit it from Beverage class.
- What if customer wants a double Mocha?

APPROACH2

- we will use decorator pattern. In this design pattern we have a component (like DarkRoast) and decorators (like Whip, Mocha, etc.). We wrap this component with decorators. This is what I mean by wrapping:

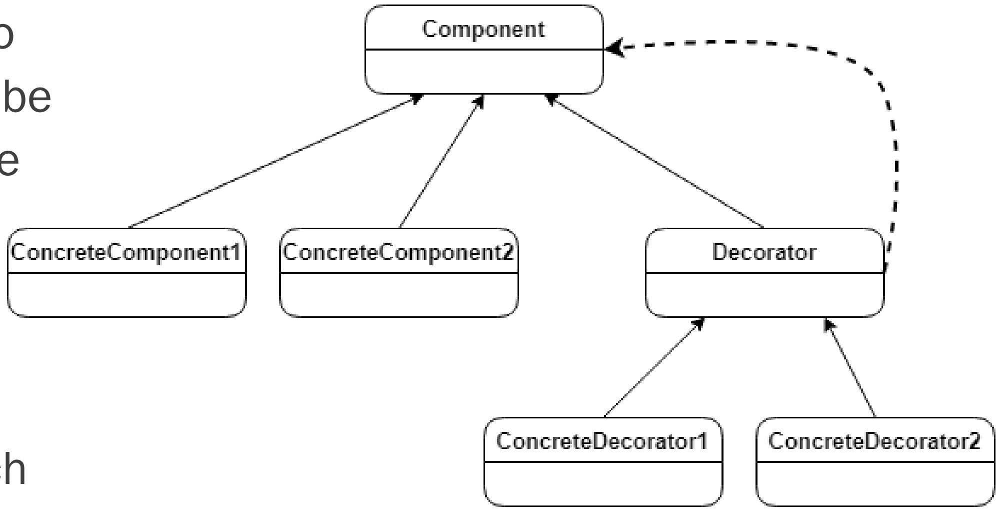


- Now, when we want to compute the cost, we will call cost method on the outermost decorator, and then it will call cost method of its inner decorator/component. This will continue, until it reaches its component (DarkRoast here). Then DarkRoast will return its own cost to Mocha, Mocha will add its own cost to it and return the total to Whip. Whip will add its own cost, and return the total out. This is the flow:
- Cost of DarkRoast = 20, Mocha = 50, and Whip = 60.



- **The Decorator pattern** attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending a functionality.
- Now that we have some ideas in our mind about how this pattern works, let's see how we will implement this type of behavior

- Component: It is a abstract class; both concrete components and Decorator abstract class will inherit its behavior.
- Concrete Component: extends Component. It is the object to which we are going to dynamically add new behavior. It can be used on its own or wrapped by a decorator. (For example, we can order DarkRoast coffee or DarkRoast with Mocha.)
- Decorator: It also extends component and is an abstract class.
- Concrete Decorator: Each decorator has a component, which means that it has an instance variable that holds reference to a component (the dotted line represents this). It extends Decorator abstract class, like Milk, Soy, etc.



- Now let’s see our abstract Component class, in our case Beverage.java file:

```
public abstract class Beverage {
  String desc = "unknown beverage";
  public String getDescription() {
    return desc;
  }
  public abstract int cost();
}
```

- The cost method is abstract because we want all of the classes that inherit it (concrete components and decorators) to override it and provide their own cost.
- Now let’s see one of our concrete component class:

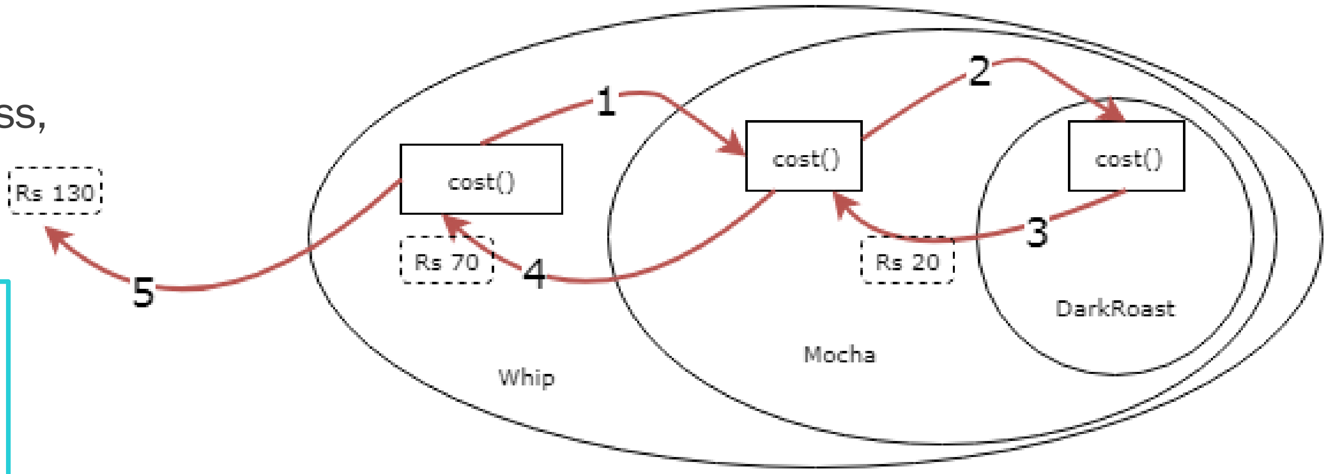
```
public class DarkRoast extends Beverage {
  public DarkRoast() {
    desc = "Dark roast";
  }
  @Override
  public int cost() {
    return 20;    }    }
```

- Whenever we define a concrete component, we will give it description in constructor and override cost method and return its cost.
- Now it’s time to see the main class where magic happens. Let’s see the abstract decorator class; in our case, it’s CondimentDecorator.java

```
public abstract class CondimentDecorator extends Beverage
{
    Beverage beverage;
    public abstract String getDescription();
}
```

- Note that it also extends Beverage. The getDescription method is abstract because we want all of the concrete decorators to override it and provide their own implementation for it. As we have not implemented cost method here, every concrete decorator will have to override it also. Additionally, we have a instance variable beverage because we will require it in concrete decorator class. Only because of this will we be able to form a chain of call to parent's cost method like this:

- Let's have a look at one of the concrete decorator class, Mocha.java



```
public class Mocha extends CondimentDecorator {
  public Mocha(Beverage b) {
    beverage = b;
  }
  @Override
    public String getDescription() {
      return beverage.getDescription() + ", Mocha";
    }
  @Override
  public int cost() {
    return beverage.cost() + 50;
  }   }
```

- In the constructor, it will receive a Beverage and set its instance variable to it and override the methods. Note that the beverage variable is inherited from the CondimentDecorator class.
- Before ordering coffee, remember that both decorators (Whip, Soy) and components (DarkRoast, Decaf) extends component (Beverage) class. So we can say that:
- DarkRoast is a Beverage.
- DarkRoast with Mocha is also a Beverage.
- DarkRoast with Mocha and Whip is also a Beverage.
- Now let's order some coffee! ☐ In your main method, test this code:

```
Beverage b1 = new Espresso();
System.out.println(b1.getDescription() + " Rs." + b1.cost());
// ** OUTPUT **
// Espresso Rs.30
```

- Ordering a simple coffee is easy and simple; let’s add a condiment.

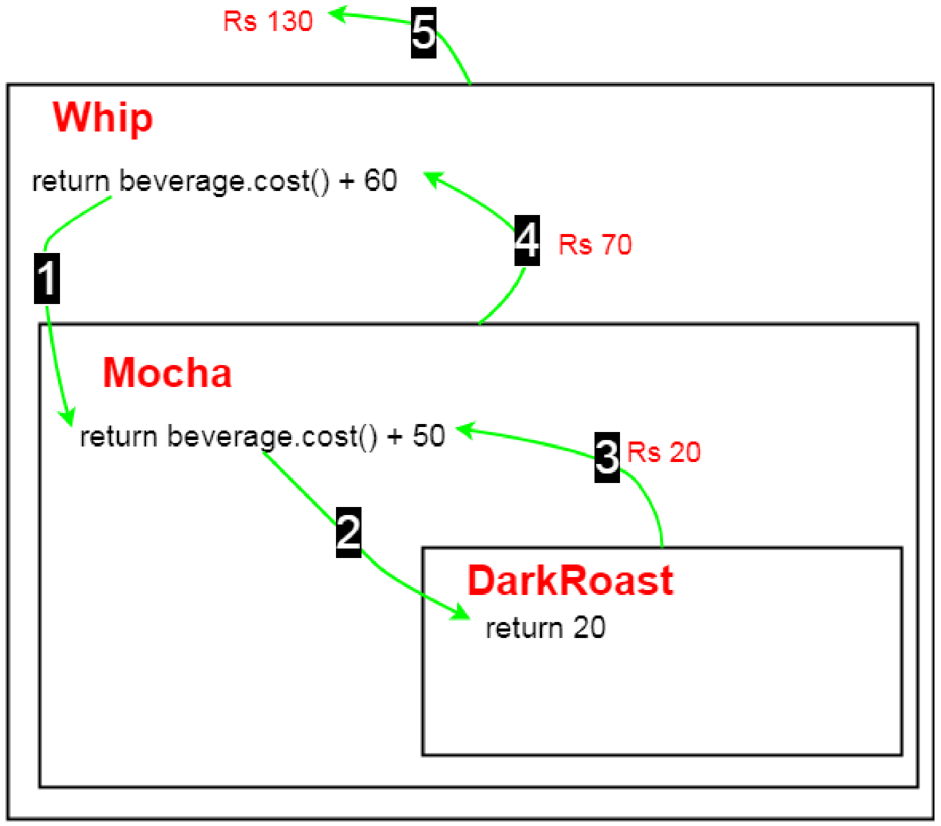
```
Beverage b2 = new DarkRoast();
b2 = new Mocha(b2);
System.out.println(b2.getDescription() + " Rs." + b2.cost());
// ** OUTPUT **
// Dark roast, Mocha Rs.70
// Rs 70 = Rs 20(of DarkRoast) + Rs 50(of Mocha)
```

- b2 = new Mocha(b2). This is what we mean by decorating a object. We have taken a object of DarkRoast and decorated with the Mocha class. Remember that the concrete decorator (Mocha here) class receives a beverage in its constructor? Here we have passed b2. So you can say that in the Mocha class beverage = b2, and at this time b2 was an object of theDarkRoast class. When we compute the cost, beverage.cost will return the cost of b2 (DarkRoast = Rs 20), and then we add 50 in it, so the result is Rs 70.

- Now let’s add two condiments, Mocha and Whip:

```
Beverage b2 = new DarkRoast();
b2 = new Mocha(b2); // wrap it with mocha
b2 = new Whip(b2); // wrap it with whip
System.out.println(b2.getDescription() + " Rs." + b2.cost());
// ** OUTPUT **
// Dark roast, Mocha, Whip Rs.130
// Rs 130 = Rs 20(of DarkRoast) + Rs 50(of Mocha) + Rs
60(of Whip)
```

- As I have already told you, (DarkRoast +Mocha) is also a beverage. This is the reason that we can pass b2 object in Whip constructor. Again, the flow is similar as before. When we call b2.cost(), first it will compute beverage.cost(), and for Whip, beverage is DarkRoast + Mocha. When we call beverage.cost() on this DarkRoast + Mocha, it will call its beverage.cost(), and for (DarkRoast + Mocha) system, beverage is DarkRoast. Here beverage.cost() will give us 20; we add 50 (Mocha cost) to it and return 70. This 70 is received by Whip, and it adds 60 and returns 130.



Note

- Last important point to note is that, although Decorator is a subclass of the Component class, we can not directly make objects of concrete decorator without passing in a concrete component. For example, here we cannot directly order only Mocha because component requires a Beverage to pass in constructor. Mocha is a decorator so it must be used to decorate a Beverage. So first make a object of concrete like DarkRoast, Decaf, or some other coffee and then decorate it with decorators.

PROS & CONS

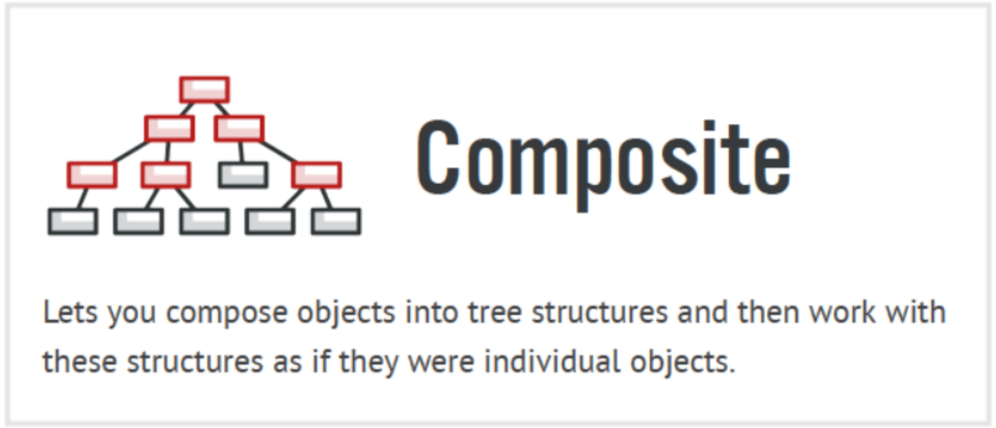
Pros

- You can extend an object’s behavior without making a new subclass.
- You can add or remove responsibilities from an object at runtime.
- You can combine several behaviors by wrapping an object into multiple decorators.

cons

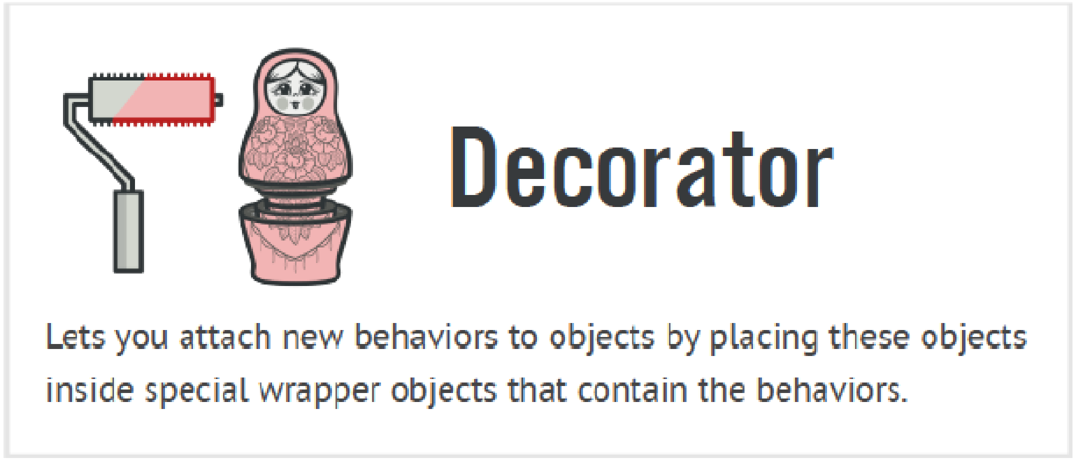
- It’s hard to remove a specific wrapper from the wrappers stack.
- The main disadvantage of decorator design pattern is code maintainability because this pattern creates lots of similar decorators which are sometimes hard to maintain and distinguish.

QUESTIONS!



Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.



Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.