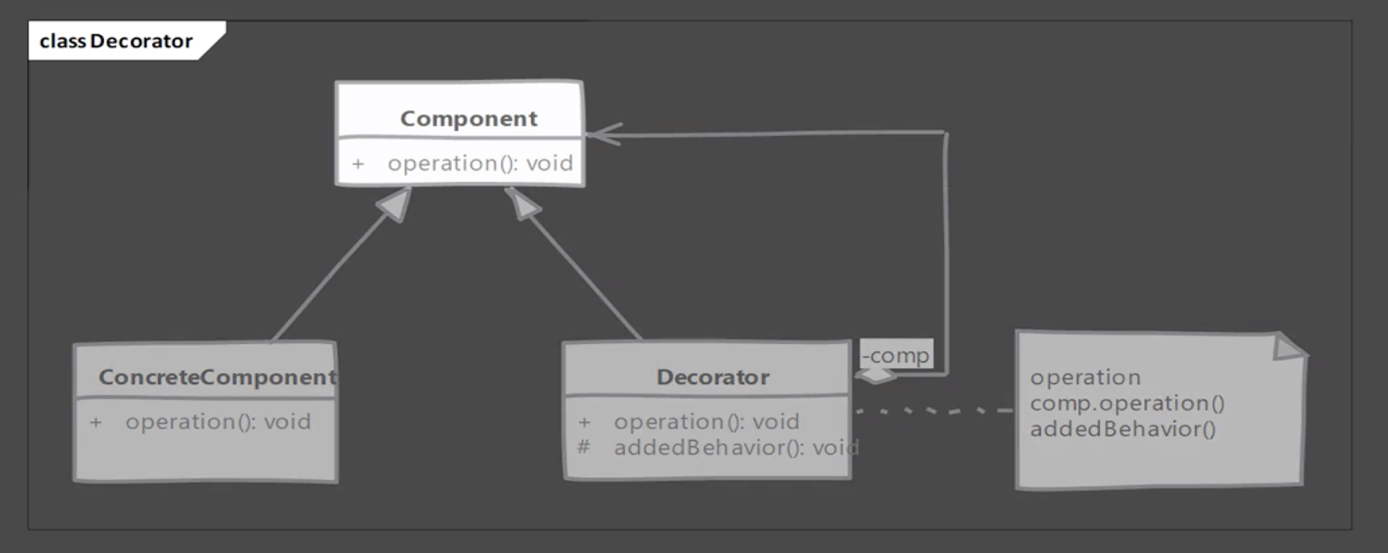
**Decorator Pattern**

* **Introduction:**
  + When we want to enhance behaviour of our existing object dynamically as and when required then we can use decorator design pattern.
  + Decorator wraps an object within itself and provides same interface as the wrapped object. So the client of original object doesn’t need to change.
  + A decorator provides alternative to sub-classing for extended functionality of existing classes.
* **UML:**

****

**A screenshot of a cell phone

Description automatically generated**

* **Implementation Steps:**
  + We start with our component.
    - Component defines interface needed or already used by client.
    - Concrete component implements the component.
    - We define our decorator. Decorator implements component & also needs reference to concrete components.
    - In decorator methods we will provide additional behaviour on top that provided by concrete component instance.
    - Decorator can be abstract as well & depends on sub-classes to provide functionality.
* **Example UML:**

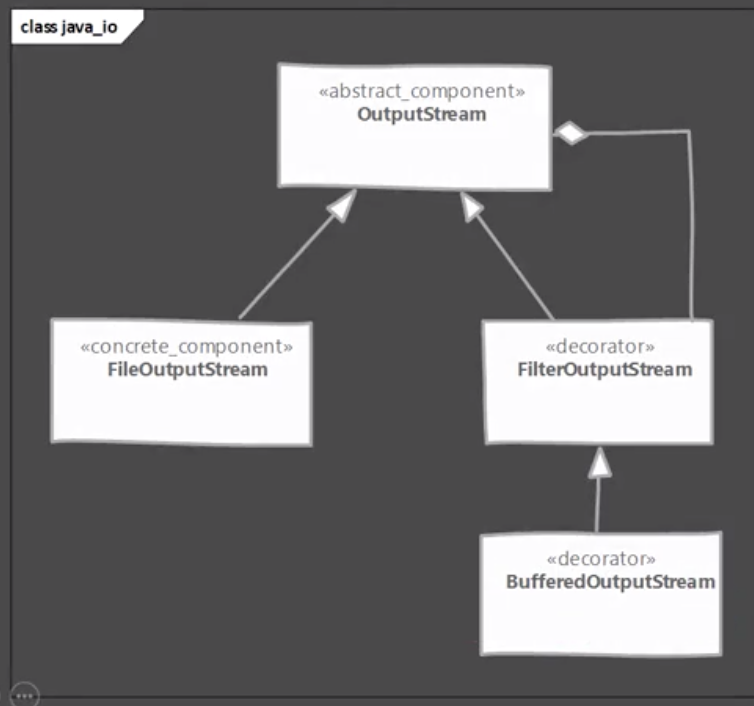
**A screenshot of a cell phone

Description automatically generated**

* **Implementation and design consideration:**
  + Since we have decorators and concrete classes extending from common component, avoid large state in this base class as decorators may not need all that state.
  + Pay attention to equals and hashCode method of decorator. When using decorators, you have to decide if decorated object is equal to same instance without decorator.
  + Decorator supports recursive composition, and so this pattern lends itself to creation of lots of small objects that add “just a little bit” functionality. Code using these objects becomes more difficult to debug.

**Design Consideration:**

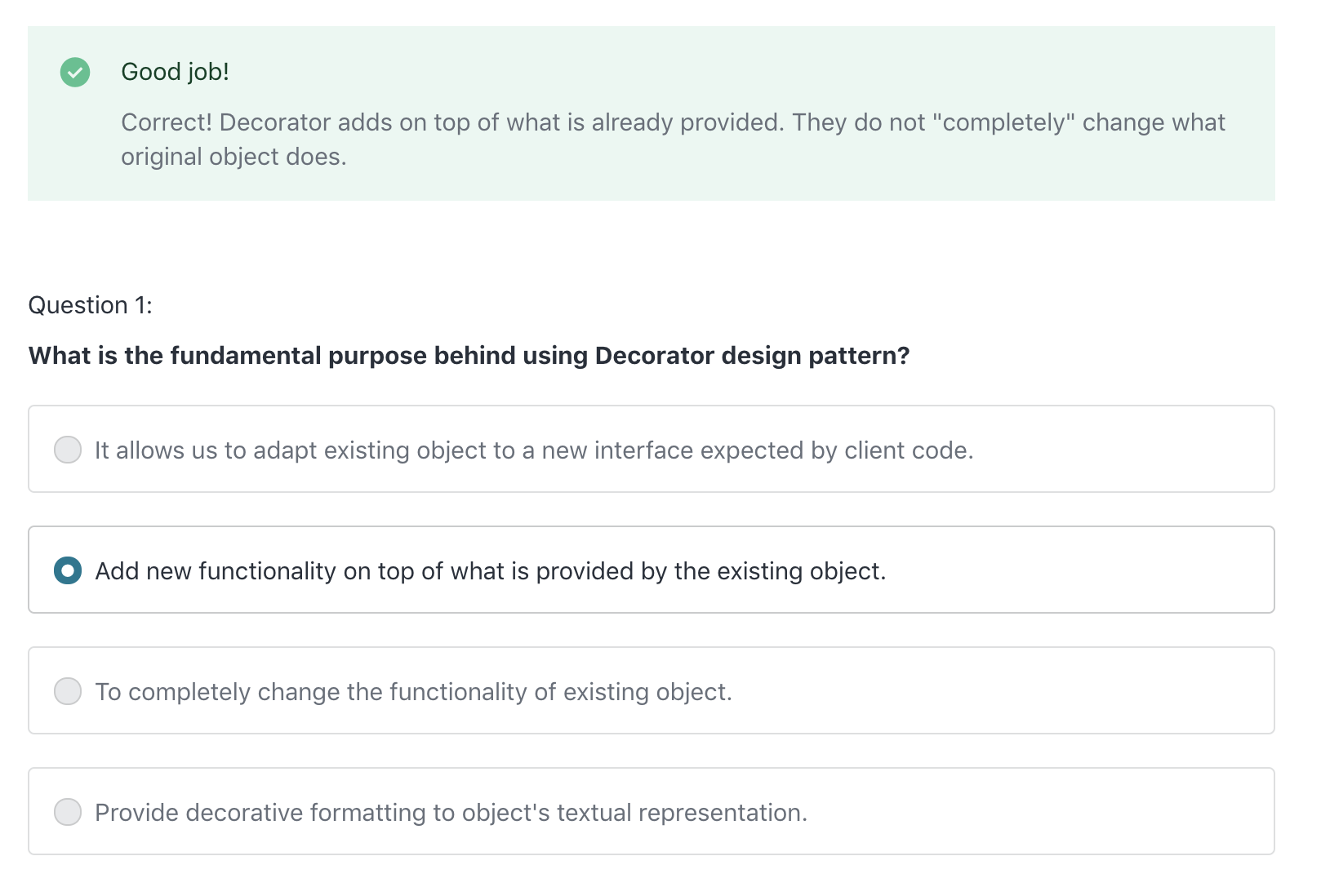
* + Decorators are more flexible & powerful than inheritance. Inheritance is static by definition but decorators allow you to dynamically compose behaviour using objects at runtime.
  + Decorators should act like additional skin over your object. They should add helpful small behaviours to object’s original behaviour. Do not change meaning of operations.
* **Example:**
  + Classes in Java I/O package are great examples of decorator pattern.
  + For example, the java.io.BufferedOutputStream class decorates any java.io.OutputStream object and adds buffering to file writing operation. This improves the disk I/O performance by reducing number of writes.

****

* **Comparison with Composite:**

|  |  |
| --- | --- |
| **Decorator** | **Composite** |
| Intent is to “add to” existing behaviour of existing object. | Composites are meant for object aggregation only. |
| Decorator can be though as degenerate composite with only one component. | Composites support any number of components in aggregation. |

* **Pitfalls:**
  + Often results in large number of classes being added to the system, where each class adds a small amount of functionality. You often end up with lots of objects, one nested inside another and so on.
  + Sometimes new comer will start using it as a replacement for inheritance in every scenario. Think of decorators as a thin skin over existing object.
* **Quiz:**

****

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**