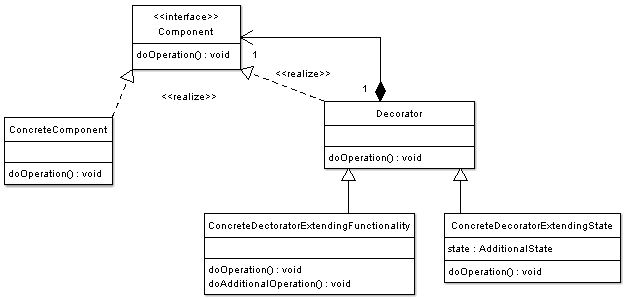
## **Decorator**

**Definition**: The intent of this pattern is to add additional responsibilities dynamically to an object. Decorators provide a flexible alternative to subclassing for extending functionality. This is also called “Wrapper”. A decorator is different from an adapter in that a decorator changes the**object’s responsibilities**, while an adapter changes an **object interface**.

**Problem**: You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class. If your application does some kind of filtering, then Decorator might be a good pattern to consider for the job.

**Solution**:



**Participants:**

* **Component** — Interface for objects that can have responsibilities added to them dynamically.
* **ConcreteComponent** — Defines an object to which additional responsibilities can be added.
* **Decorator** — Maintains a reference to a Component object and defines an interface that conforms to Component’s interface.
* **Concrete Decorators** — Concrete Decorators extend the functionality of the component by adding state or adding behavior.

**Example**:

Widget (Component): TestField (ConcreteComponent), **Decorator**

* **Decorator**: BorderDecorator (Concrete Decorator), ScrollDecorator (Concrete Decorator)
* Client (main function):

Widget \*aWidget = **new** BorderDecorator(**new** BorderDecorator(**new** ScrollDecorator(**new** TextField(80, 24))));  
aWidget->draw();