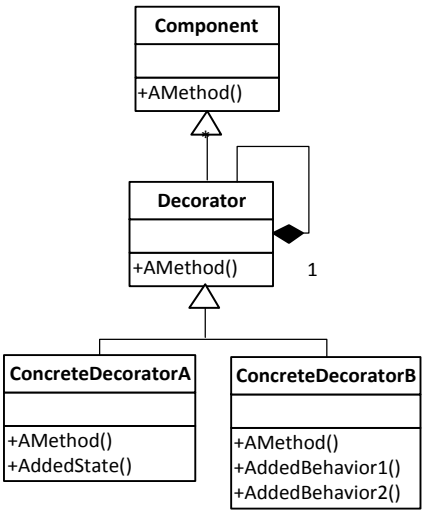
9 Decorator Pattern [Gamma et al]

Functionalities evolve from day to day based on the additional requirement on a current proposed structure. As a common rule of thumb new functionality is added either using inheritance or composition. Inheritance inserts hierarchy problems into the structure. Root Class level modifications introduce an obvious impact on all the implementors. Decorator design pattern is a solution to such problems where dynamic responsibility is handled by the decorators. Think of decorator as a skin that add some look and feel (behavioral) to an existing object advocating delegation instead of separate inheritance. The intent of decorator pattern covers the runtime functionality assigned at the scope of the object not class. [26], [34] and [35] describes an application example of decorator pattern. Following are the variants of decorator pattern.

9.1 Added responsibility to run bottom up and top down [36]

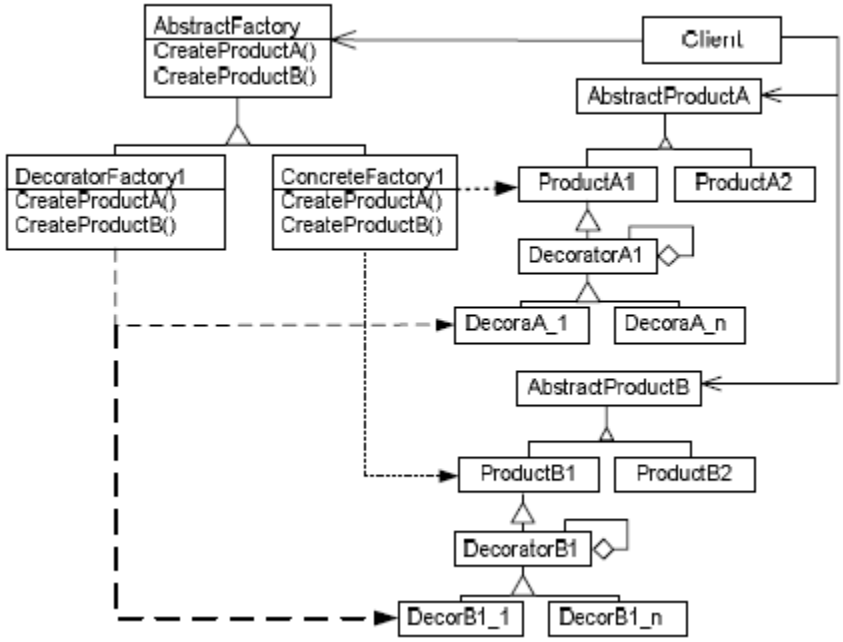
In this variant decorator pattern is modified to add testing behaviors which is used in the testing of classes. The goal was designed with state of mind that source remains unchanged and no recompilation for the overall system is to be made explicitly. New behaviors (tests) are added by the Concrete Decorators along with previous old cases which benefits the execution of tests from top to bottom i.e. from Component class to ConcreteDecoratorB class. Conversely the bottom up can be done by reversing the link of the Decorator field in the Decorator class.



9.1 UML Diagram of bottom up and top down execution of behaviors approach [36]

9.2 Decorators with factories [36]

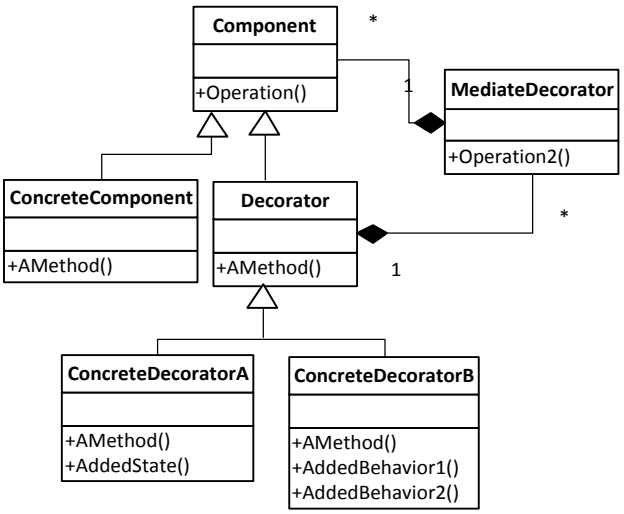
In order to restrict the creation of objects directly from “new” operator and check on their dynamic behavior, systems have different representations based on external conditions that drive the creation of Decorator. The author [36] created a test case Client which it uses it to test the hierarchy and then decouple it with no harm of source code change.



9.2 UML Diagram of decorator pattern implemented with abstract factory [36].

9.3 Mediate Decorator [37]

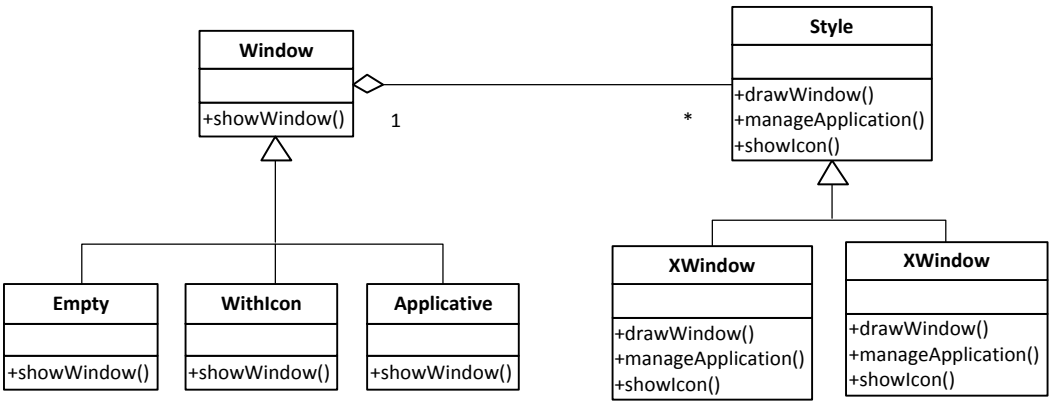
This variant describes a method where a third class that is not implemented by the Component Class invokes the operation of the main Component Class. The component class now contains a reference link of the third class called MediateDecorator which has a reference to the Decorator class. The MediateDecorator is like second component and the Decorator Class implements the first component and invokes the functionality of the second component via a composition link.



9.3 UML Diagram of Mediate Decorator pattern [37].

9.4 Delegation between abstraction and implementation [33]

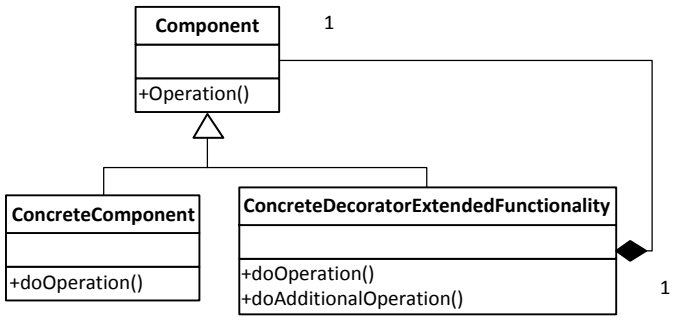
This variant is described by delegation between ConcreteComponent Classes and abstraction interface i.e. Style. The composition link carries the opportunity to invoke the abstraction operations through Window. As an example, Windows styles vary from platform to platform which involves client code to be executed independent of the platform



9.4 UML Diagram of delegation between abstraction and implementation [33].

9.5 Omitting the abstract Decorator class [1]

For sole responsibility, Decorator class can be eliminated if an existing structure already exists rather than creating from scratch. The Decorator responsibility is embedded into the ConcreteDecorator Class level.



9.5 UML Diagram with no abstract decorator class [1].