Decorator Design Pattern

Purpose and Motivation

The purpose of the Decorator pattern is to attach additional responsibilities to an object dynamically without changing how it is used by the user. In this way, decorated objects are not dependent on its previous functionalities, which is advantageous to a programmer in many ways. Using the decorator pattern, you can add additional functionality to the object, but still handle it as if it were the base object, use multiple decorators to add additional functionalities without making a subclass for every possible combination, and even decorate objects at runtime

Intended Use Cases

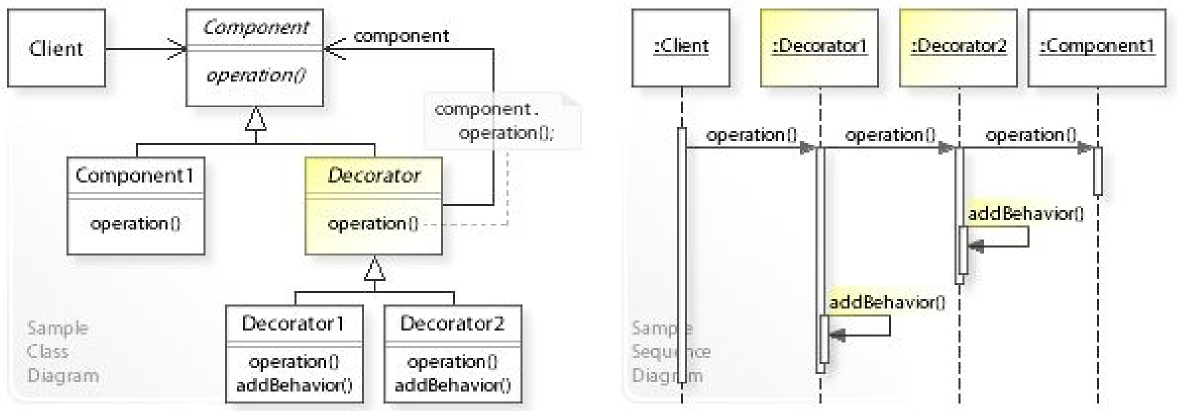
Use Decorator

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Vocabulary

- **(abstract) Component** - Abstract class or interface with an operation that we would like to decorate
- **Concrete Component** - Concrete class that implements the operation we would like to decorate
- **(abstract) Decorator** - Abstract class that holds a reference to a component, and contains functionality to perform an operation to that component.
- **Concrete decorator** - Concrete class that overrides the operation implementation to uniquely "decorate" the object.

Structure and Runtime Behaviour



Known Consequences

Positive Consequences

- Decorated objects can be used exactly the same as undecorated objects, implementing the extended functionality without changing the internal structure
- Creating a new decorator does not require the knowledge of how other decorators are implemented, which is extremely useful when working in a team.
- Any number of decorators can be applied to a single component, easily creating any combination of extended functionality, as opposed to extending the class for each combination, which may lead to a large amount of class definitions.
- The decorator pattern works at runtime, as opposed to inheritance, which is done at compile time

Negative Consequences

- The decorator pattern may lead to numerous small classes that only differ in one or two properties

NFPs

Improved NFPs

Scalability: the decorator pattern is designed to be extremely scalable as adding new functionality to any object can be done by simply adding a new decorator the implements the desired operation.

Inhibited NFPs

Extensibility: new functionality can be added to (or removed from) objects dynamically at runtime.

Credits

Students:

IGAME (Yuzhan Jiang, Zihan Liang, Yinglun Suo, Wanghao Tang)

Puzzle (Dhivagar Gnanaratnam, Kyung Keum, Anthony Tu, Henry Zhu)

Instructor and TAs:

Mei Nagappan, Arman Naeimian, Ivens Portugal