



Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

OO Principles

Encapsulate what varies.
 Favor composition over inheritance.
 Program to interfaces, not implementations.
 Strive for loosely coupled designs between objects that interact.
 Classes should be open for extension but closed for modification.

OO Basics

action
 isolation
 morphism
 chance

We now have the Open-Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

OO Patterns

Strategy
 encapsulate
 interface
 vary

Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

And here's our first pattern for creating designs that satisfy the Open-Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component—that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.