# 11. Adding new functionality to existing objects that cannot be modified

Imagine that you have the following situation. There is either some third party library, or your own legacy code that you need to use. And you need to make some changes to its functionality. You would either need to modify the existing behavior, or add some new behavior. But you can't change the components that you are about to use. This is because you either don't have access to the internal code of those components, or simply aren't allowed to make this change.

Or maybe the external component is not fully compatible with your code, so you will need to both make it compatible and then extend it. However, making the component compatible with your code is a different problem that we have already covered. The specific problem that we are looking at now is the ability to extend the behavior of the objects that you can't modify directly.

## Suitable design patterns

### Decorator

Decorator is a design pattern that is similar to Adapter. Just like with Adapter, the Decorator class acts as a wrapper around the original object. However, instead of changing the interface of the

original object, it uses the same interface as the existing object. The API will be exactly the same.

You can think of a silencer on a pistol to be analogous to the Decorator design pattern. The original pistol already has a specific configuration that you cannot change. But attaching a silencer to it makes it behave differently. While the silencer is attached, however, the original pistol still remains intact. The silencer can be removed at any point.

So, when you apply the Decorator design pattern, anything in your code that could previously use the original object would still be able to use the Decorator class in its place. And this is precisely because the original structure of the interface was left intact.

### Why would you use Decorator

- It will give you the ability to easily add functionality to those objects that you can't modify directly.
- Open-closed principle is enforced.
- You can use it recursively by applying additional decorators on top of the existing ones.
- You can dynamically add responsibilities to an object at runtime.
- The single responsibility principle is applied well, as each decorator can be made responsible for only a single enhanced functionality