# Open for extension but closed for modification

* Our goal is to allow classes to be easily extended to incorporate new behaviour without modifying existing code.
* This enables our code to be resilient to change and flexible enough to take on new functionality to meet changing requirements.

# Decorator Pattern

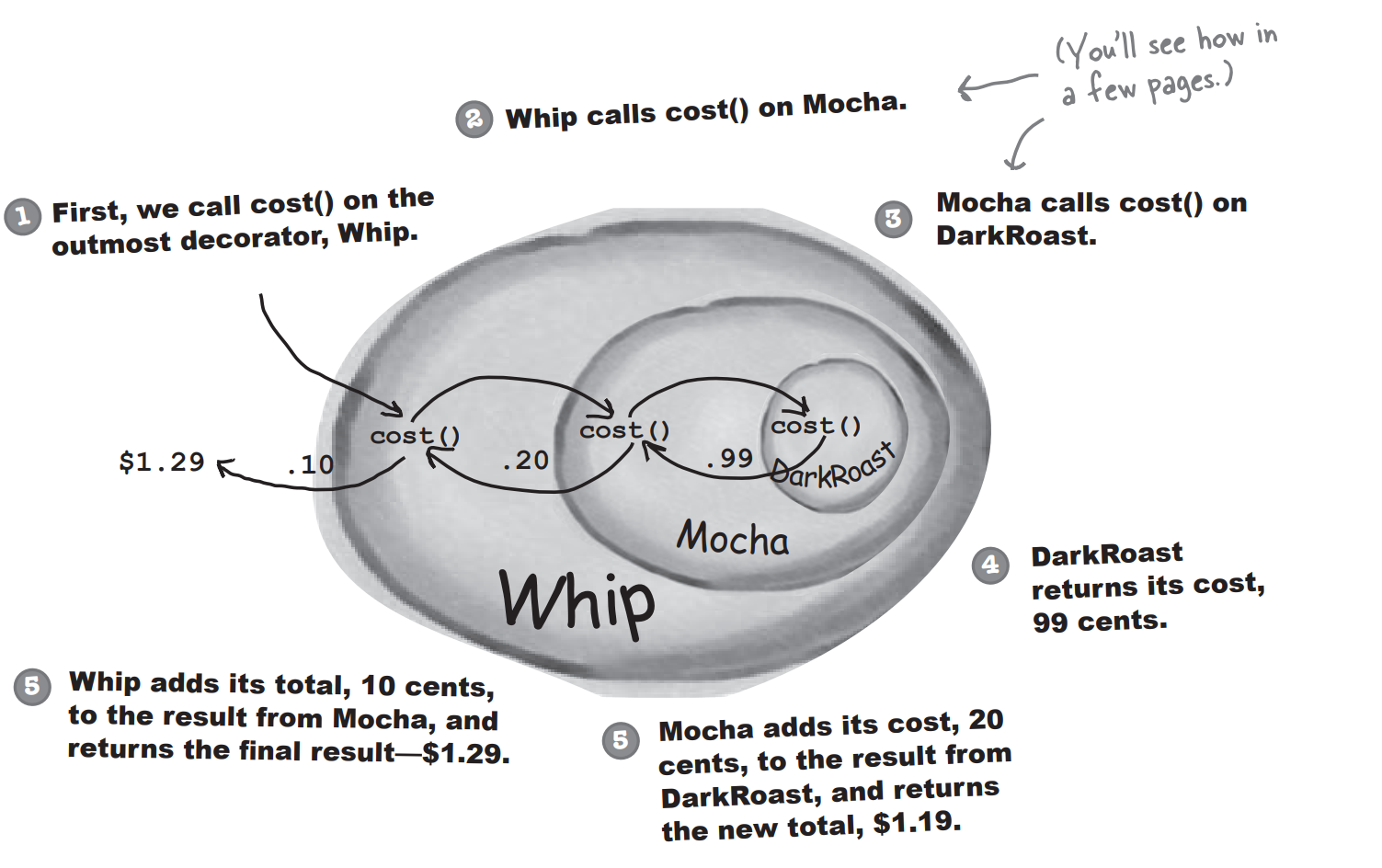
* Instead of overusing inheritance, we can **decorate our classes at runtime using a form of object composition.**

🡪 Once learning decorator patterns, we will be able to give object new responsibilities without making any code changes to the underlying classes.

* The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorator provide a flexible alternative to subclassing for extending functionality.

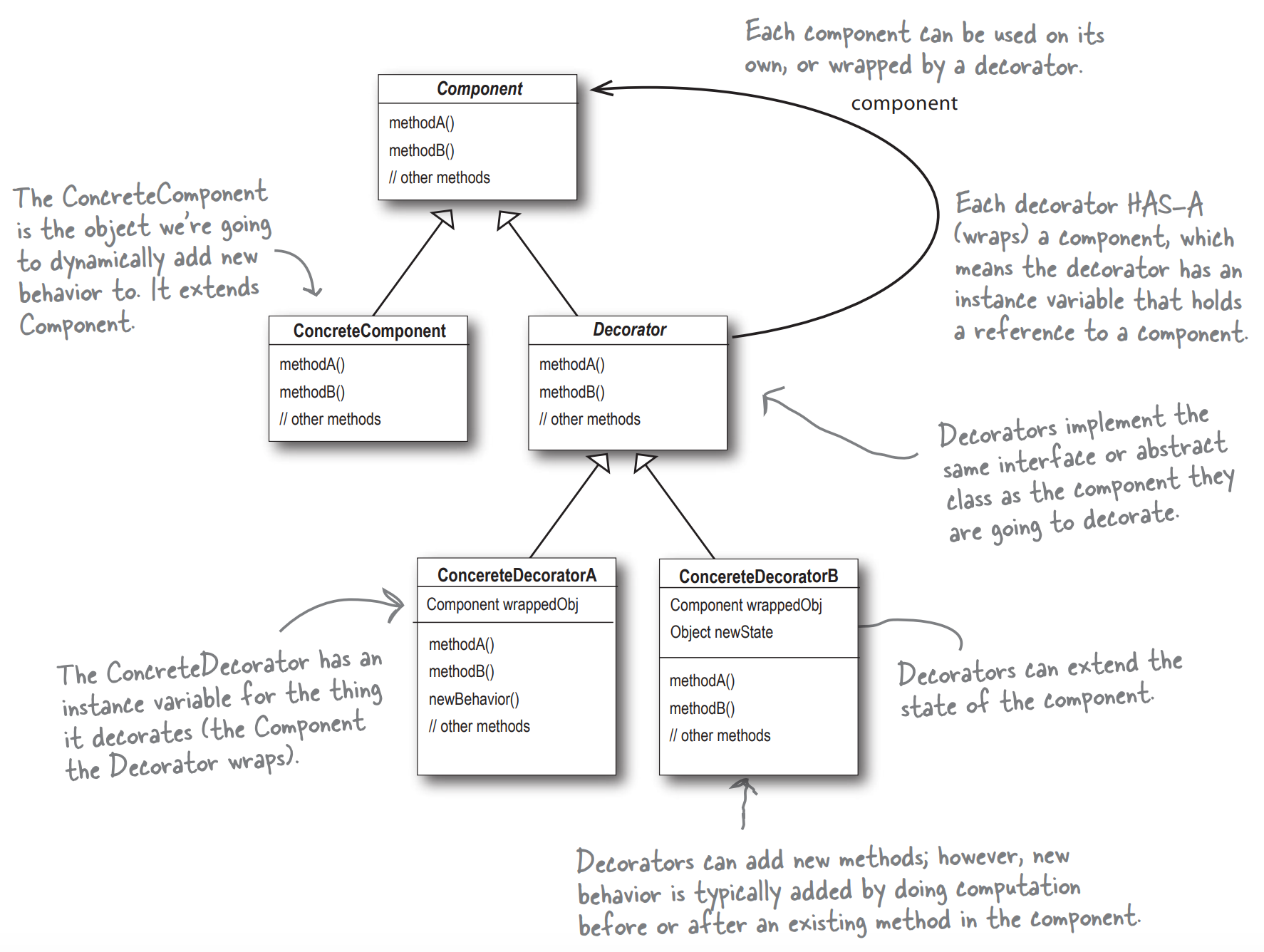
## Constructing a drink order with Decorators

* Compute the cost for the customer.
* We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects in decorates.
* Once it gets a cost, it will add on the cost of the Whip.



|  |
| --- |
| **Okay, Here’s What we know so far** |
| * Decorators have the **same supertype** as the objects they decorate. * You can use one or more decorators to wrap an object * Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object * **The decorator adds its own behaviour** either before and/or after delegating to the object it decorates to do the rest of the job. |

## The Decorator Pattern – Class Diagram



# Decorating our Beverage../../../Library/Group%20Containers/group.com.apple.notes/Media/61552AF3-761D-42A0-81EE-6E2C13DD52FB/Fullscreen_21_7_18__12_44_pm.png