# Software Design Patterns

*Lecture 7*

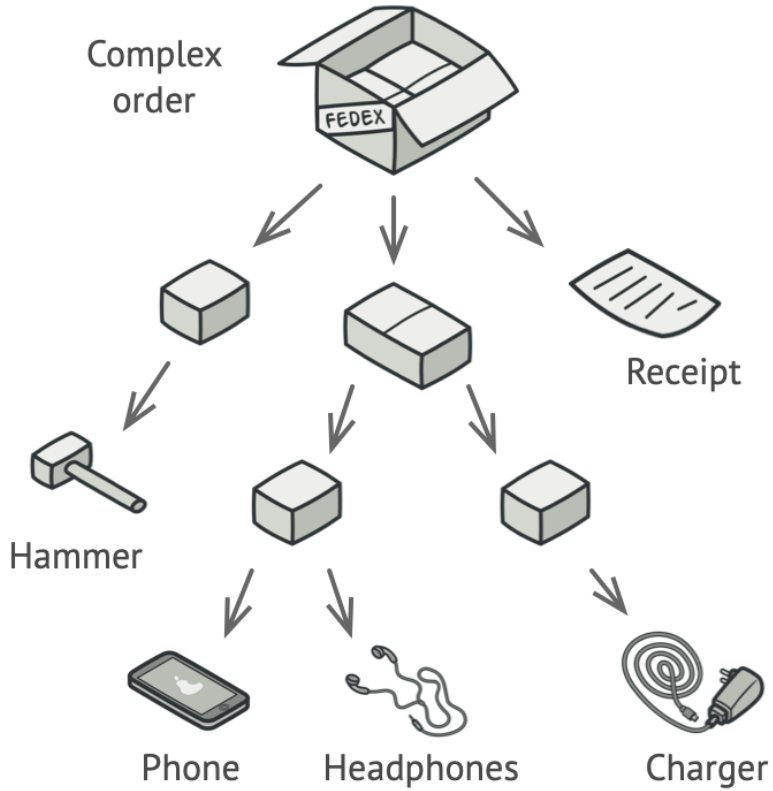
## *Composite, Decorator, and Facade*

**Dr. Fan Hongfei**

**17 October 2024**

# Composite: Problem

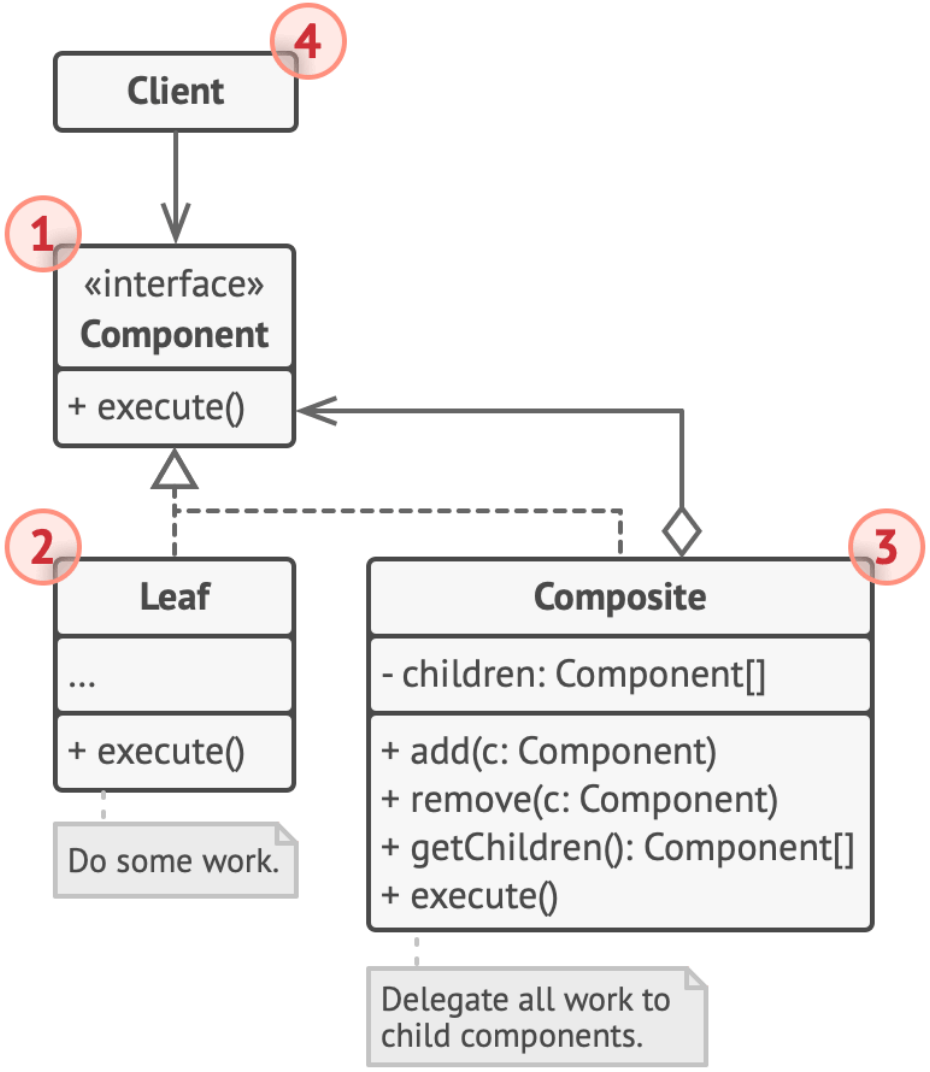
- **Example**
  - Two types of objects: Products and Boxes
  - Requirement: calculating the total price of an order
- **A direct approach**
  - Unwrap all boxes and go over all products
  - Problem: not as simple as running a loop




[Figures in this slide are extracted from https://refactoring.guru/design-patterns/composite]

# Composite: Solution and Structure

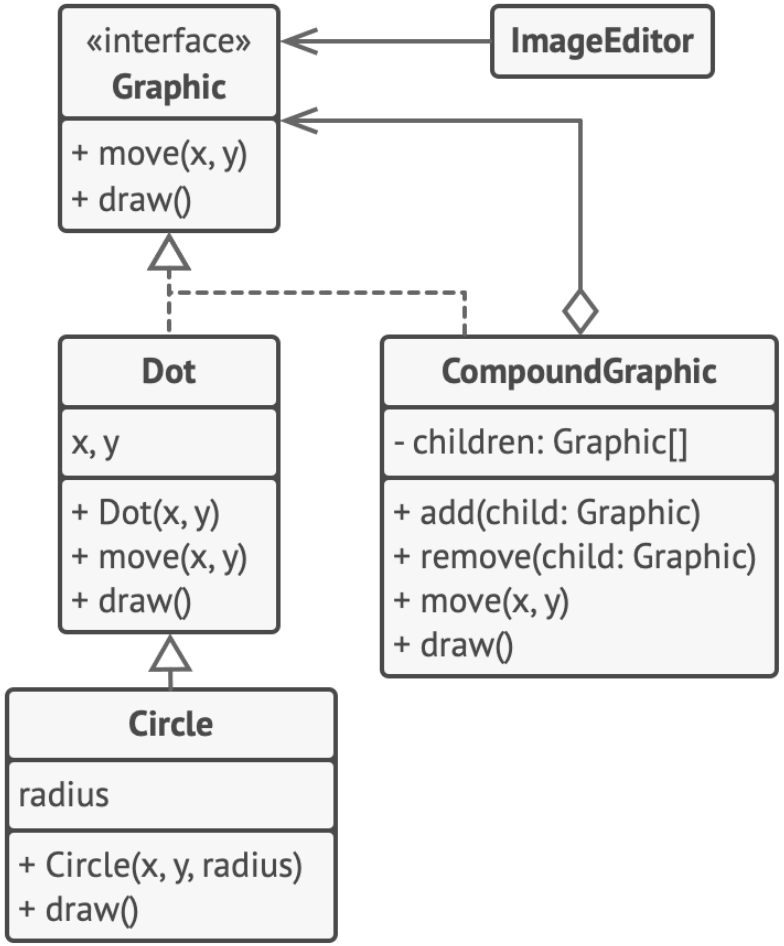
- **Composite (aka Object Tree)**
- Working with **both** Products and Boxes through **a common interface** that declares a method for calculating the price
  - For a product: simply returning the price
  - For a box: **go over each item** it contains, and may even **add extra cost**
- **Benefit**
  - There is no need to care about concrete classes of objects that compose the tree
  - Treat all items via the common interface




[Figures in this slide are extracted from https://refactoring.guru/design-patterns/composite]

# Composite: Example

- Implementing stacking of geometric shapes in a graphical editor




- The CompoundGraphic class is a container that can comprise any number of sub-shapes
- A compound shape passes the request recursively to all children and "sums up" the result
- The client code works with all shapes through the single interface

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/composite]

# Composite: Applicability

- When you have to implement a **tree-like** object structure
  - Two basic element types that share a common interface: simple leaves and complex containers
  - A container can be composed of both leaves and other containers
- When you want the client code to **treat both simple and complex elements uniformly**
  - All elements defined by the Composite pattern share a common interface

# Composite: Implementation

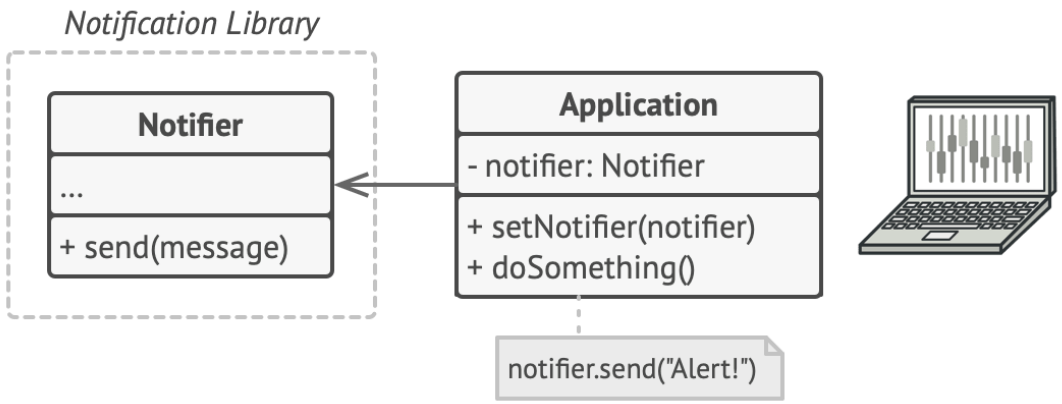
1. Make sure that the core model of the app can be **represented as a tree structure**, and break it down into **simple elements and containers**
2. Declare the **component interface** with a list of methods that **make sense for both** simple and complex components
3. Create a **leaf class** (or multiple leaf classes) to represent simple element(s)
4. Create a **container class** to represent complex elements
   - Use an array field for storing references to sub-elements: must be able to store both leaves and containers, declared with the component interface type
   - While implementing the methods, the container delegates most of the work to sub-elements
5. Define the methods for adding/removal of child elements in the container
   - NOTE: declaring these operations in the component interface would violate the Interface Segregation Principle

# Composite: Pros and Cons

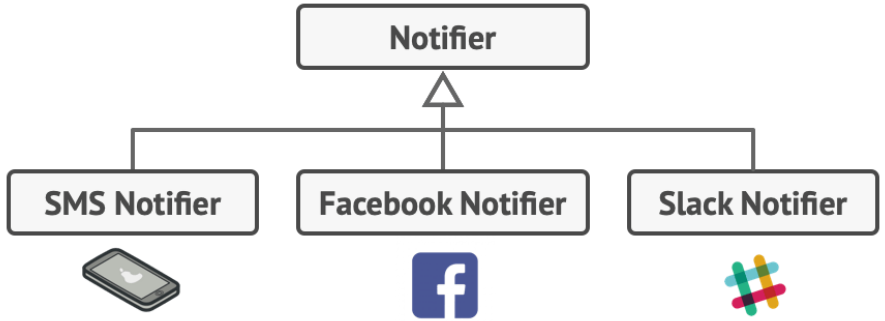
- **Pros**
  - Work with complex tree structures more conveniently: use polymorphism and recursion
  - Open/Closed Principle: introduce new element types into the app without breaking the existing code
- **Cons**
  - Might be difficult to provide a common interface for classes whose functionality differs too much, and there is a need to overgeneralize the component interface

# Decorator: Problem

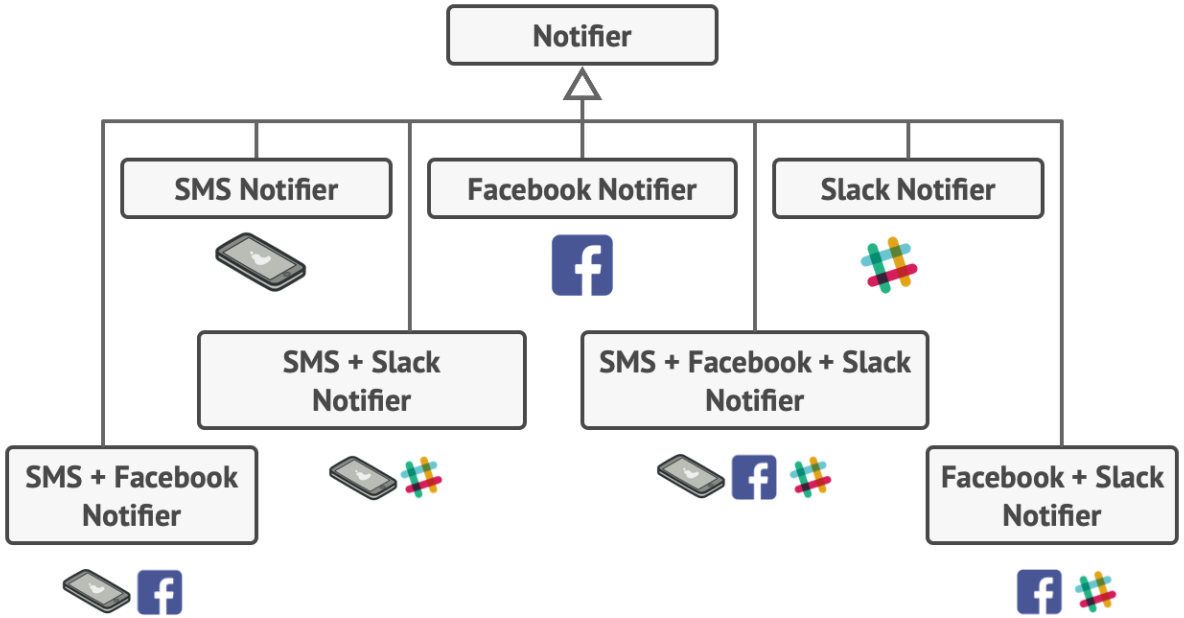
- **Example: a notification library which lets other programs notify users about important events**
- Initial version




- New requirement




- Eventually




[Figures in this slide are extracted from https://refactoring.guru/design-patterns/decorator]

# Decorator: Solution

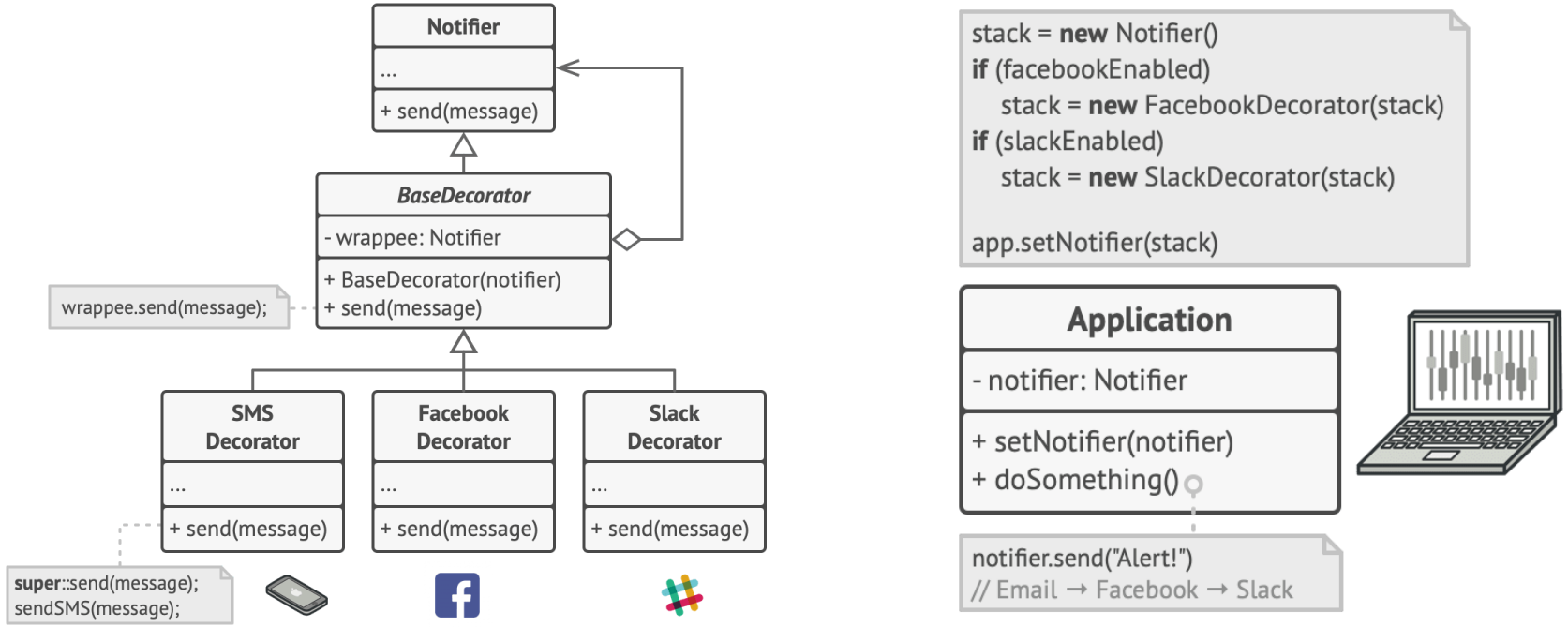
- **Problems with inheritance**
  - Inheritance is static: you can never alter the behavior of an existing object at runtime
  - Subclasses can have only one parent class
- **Solution**
  - Using Aggregation or Composition instead of Inheritance
  - Idea: one object has a reference to another, and delegates it some work


- **With such approach**
  - Easily substitute the linked "helper" object with another, changing the behavior of the container at runtime
  - An object can use the behavior of various classes

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/decorator]

# Decorator: Solution (cont.)

- **Alternative name of Decorator: Wrapper**
  - Wrapper: an object that can be linked with some target objects
  - The wrapper contains the same set of methods as the target, and delegates all requests to the target
  - The wrapper may alter the result either before or after the delegation
- **When does a simple wrapper become a decorator?**
  - **The wrapper implements the same interface as the wrapped object**



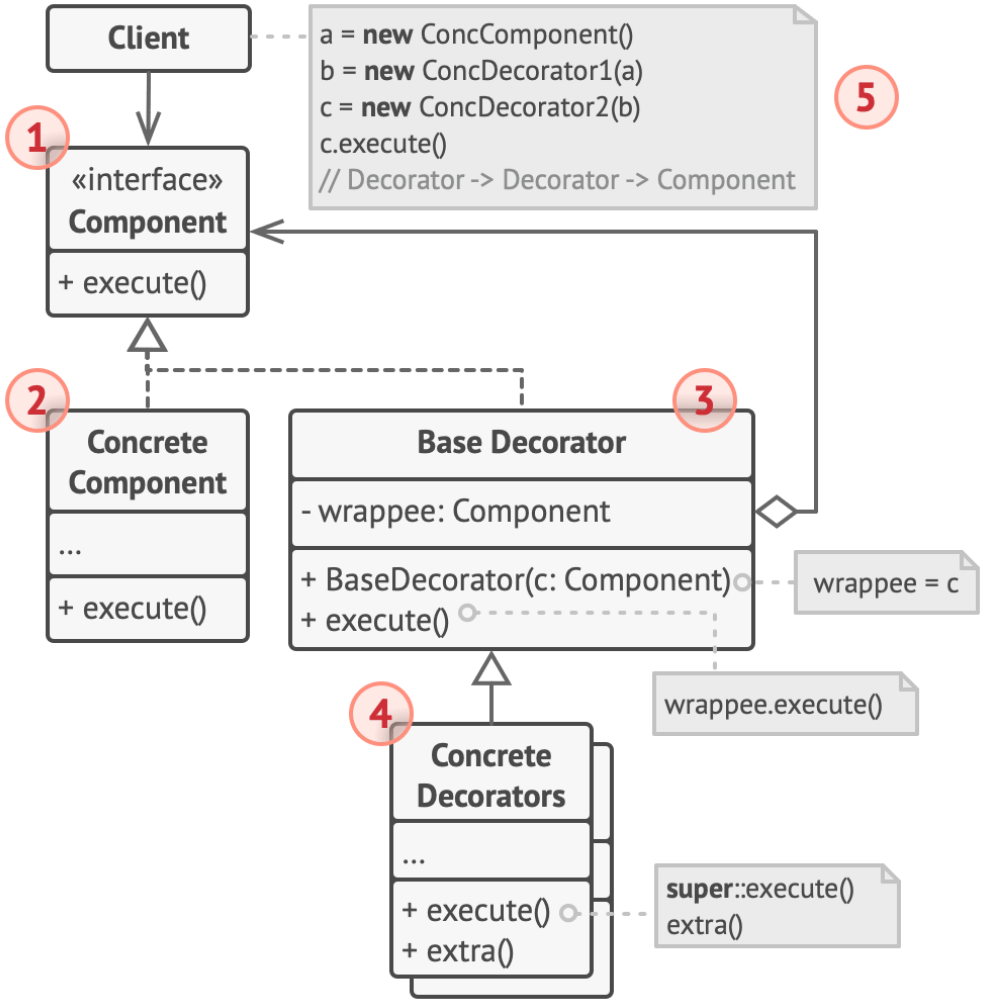


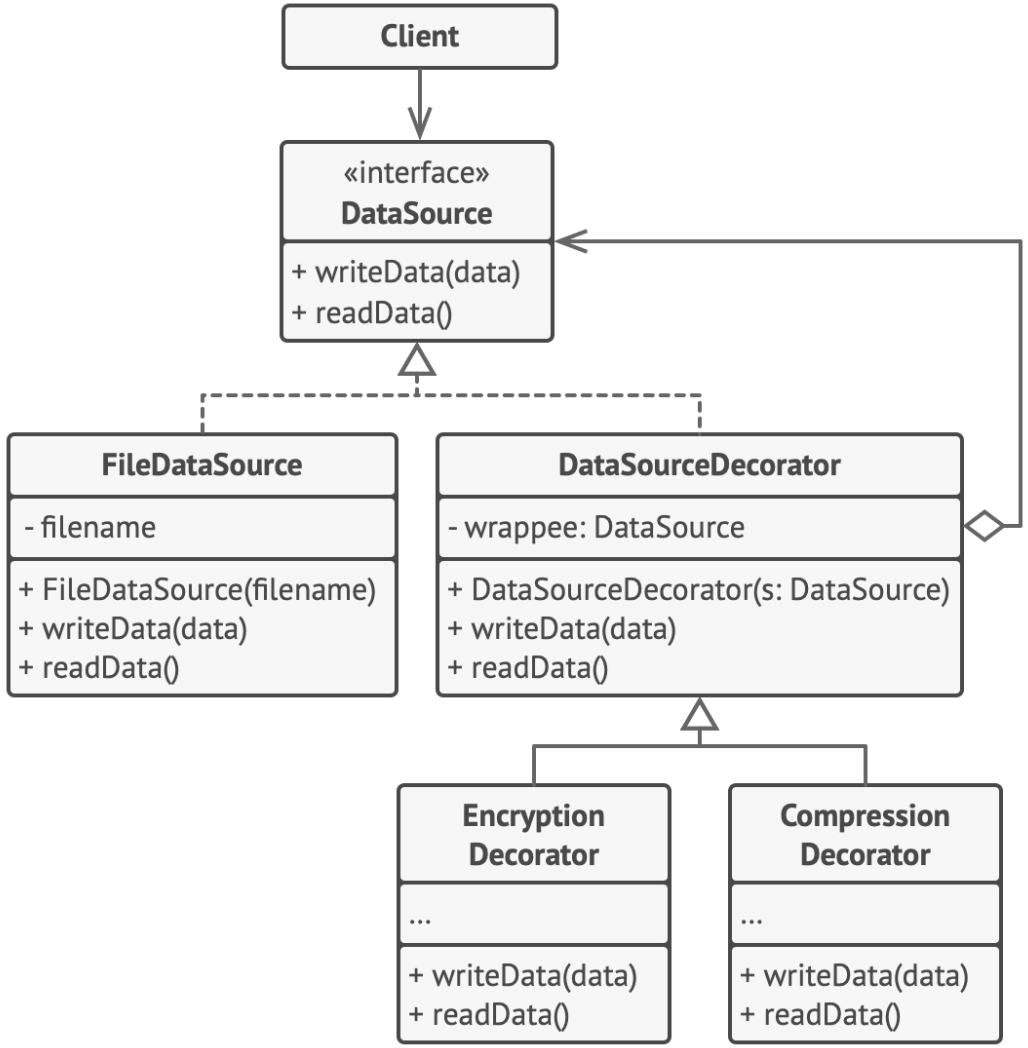


- Cover an object in **multiple wrappers**, adding the combined behavior
- The **last decorator** in the stack would be the object that the client actually works with

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/decorator]

# Decorator: Structure and Example




[Figures in this slide are extracted from https://refactoring.guru/design-patterns/decorator]

# Decorator: Applicability

- To assign extra behaviors to objects at runtime, without breaking the client code
  - The Decorator structures the business logic into **layers**
  - Create **a decorator for each layer** and compose objects with various combinations of this logic at runtime
  - The client code treats all objects in the same way, since they follow a common interface
- When it is **awkward or not possible to extend an object's behavior using inheritance**
  - For example, many languages have the final keyword to prevent further extension

# Decorator: Implementation

1. Make sure that the business domain can be represented as **a primary component with multiple optional layers over it**
2. Figure out **what methods are common** to both the primary component and the optional layers
   - Create a component interface and declare those methods
3. Create a **concrete component class** and define the **base behavior** in it
4. Create a **base decorator class**
   - Containing **a field for storing a reference to a wrapped object**, declared with the component interface type
   - Delegating all work to the wrapped object
5. Make sure all classes implement the component interface
6. Create **concrete decorators** by extending them from the base decorator
   - Execute **its behavior before or after** the call to the parent method
7. The client code must be responsible for creating decorators and composing them

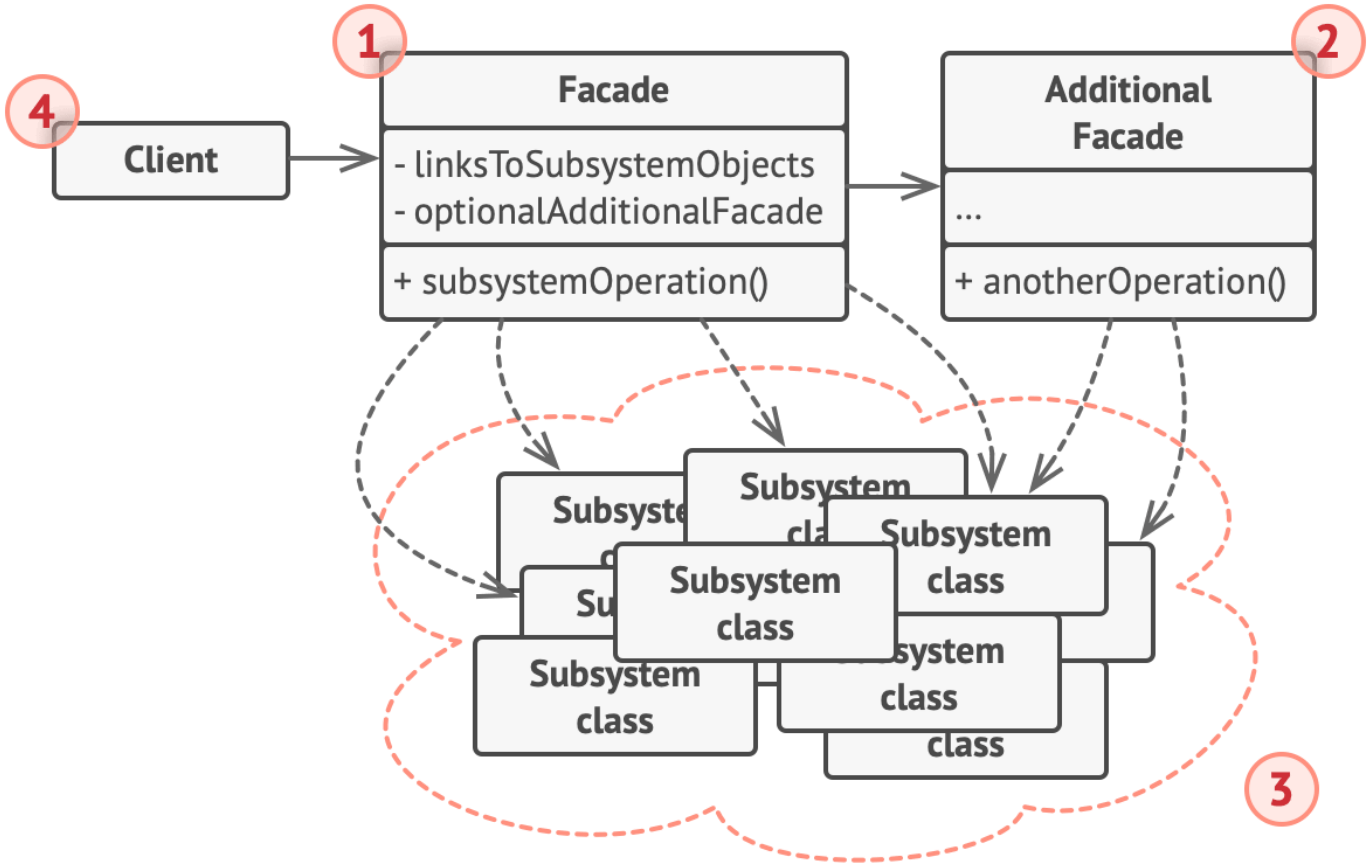
# Decorator: Pros and Cons

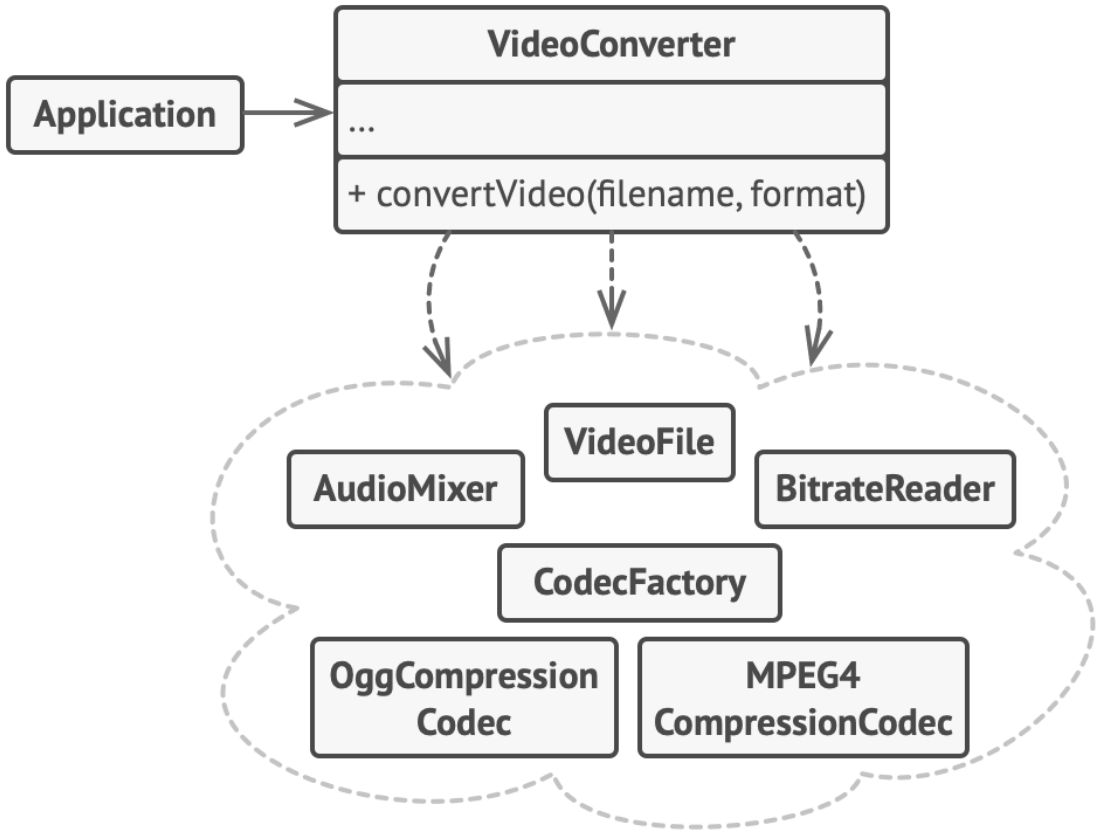
- **Pros**
  - Extend an object's behavior without making a new subclass
  - Add/remove responsibilities from an object at runtime
  - Combine several behaviors by wrapping an object into multiple decorators
  - Single Responsibility Principle: divide a monolithic class that implements many possible variants of behavior into smaller classes
- **Cons**
  - It is hard to remove a specific wrapper from the wrappers stack
  - It is hard to implement a decorator in such a way that its behavior does not depend on the order in the decorators stack
  - The initial configuration code of layers might look pretty ugly

# Facade: Problem and Solution

- **Problem: working with a broad set of objects that belong to a sophisticated library or framework**
  - Need to initialize all objects, keep track of dependencies, execute methods in the correct order
  - Result:  the business logic would become tightly coupled to the details of 3rd-party classes
- **Solution**
  - **Facade:** a simple interface to a complex subsystem which contains lots of moving parts
  - Providing **limited functionality**, only features that clients **really care about**
  - For example: an app that uploads short funny videos may use a professional video conversion library, but all it really needs is **a class with encode(filename, format)**
  - This class is a **facade**

# Facade: Structure and Example




[Figures in this slide are extracted from https://refactoring.guru/design-patterns/facade]

# Facade: Applicability

- To have a limited but straightforward interface to a complex subsystem
  - The Facade provides a shortcut to the most-used features of the subsystem which fit most requirements
- To structure a subsystem into layers
  - Create facades to define entry points to each level of a subsystem: reducing coupling between subsystems by requiring them to communicate only through facades
  - For example, the video conversion framework can be broken down into two layers: video- and audio-related

# Facade: Implementation

1. Check whether it is possible to provide a simpler interface than what an existing subsystem already provides
2. Declare and implement this interface in a new facade class
   - **Redirecting the calls** from the client code to appropriate objects of the subsystem
   - Be responsible for **initializing** the subsystem and **managing** its life cycle, unless the client code already does this
3. Make all the client code communicate with the subsystem only via the facade
4. If the facade becomes too big, consider extracting part of its behavior to a new, **refined** facade class

# Facade: Pros and Cons

- **Pros**
  - Isolate your code from the complexity of a subsystem
- **Cons**
  - A facade can become "a god object" coupled to all classes of an app