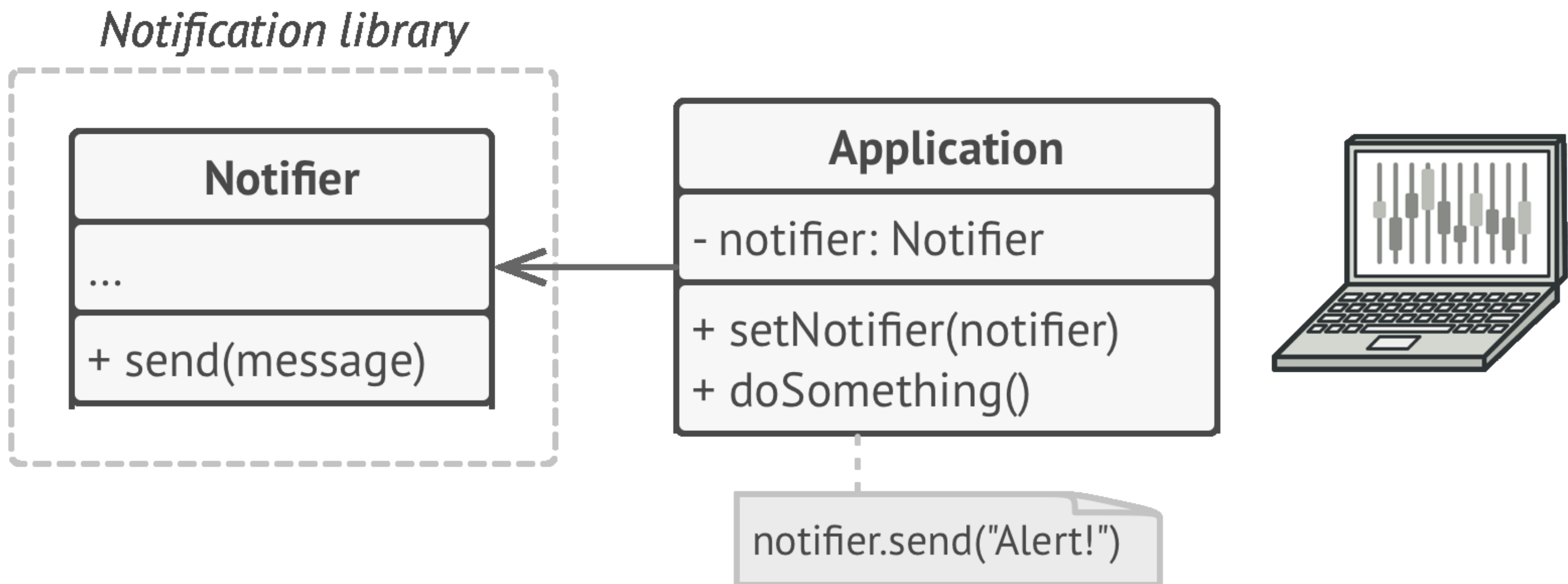
Decorator

Also known as: Wrapper

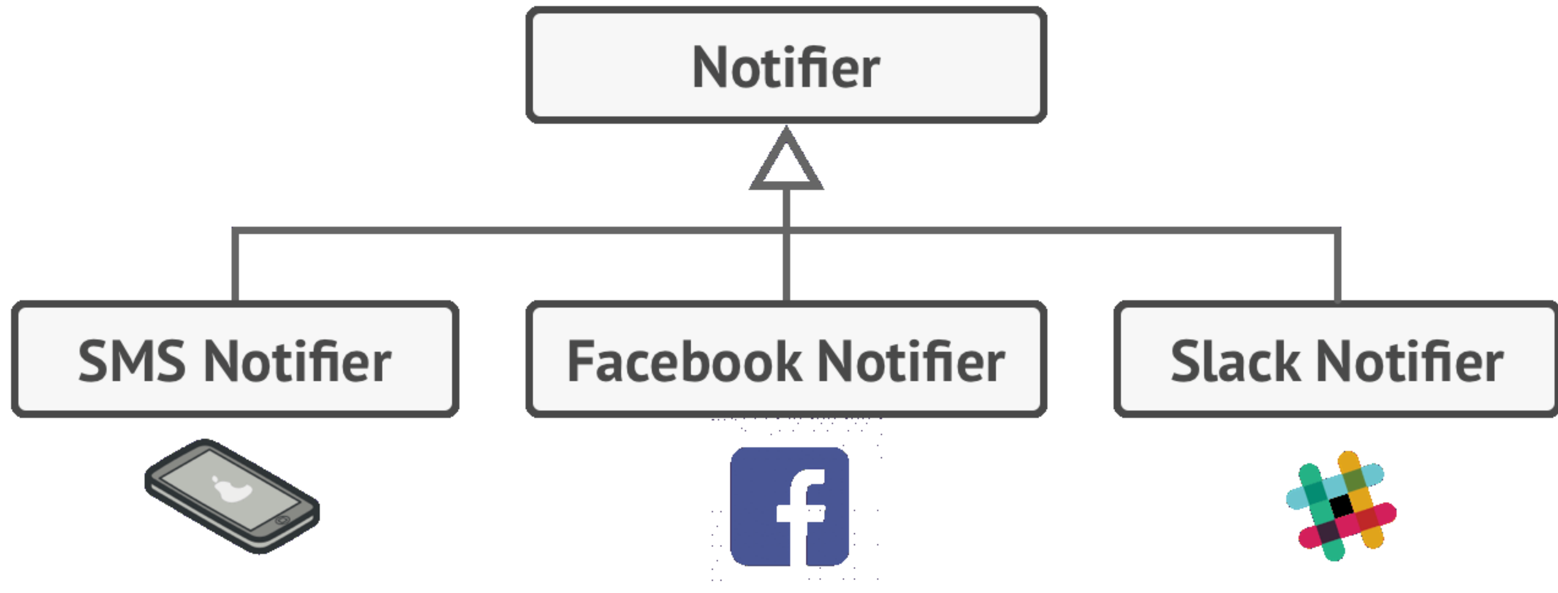
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Problem



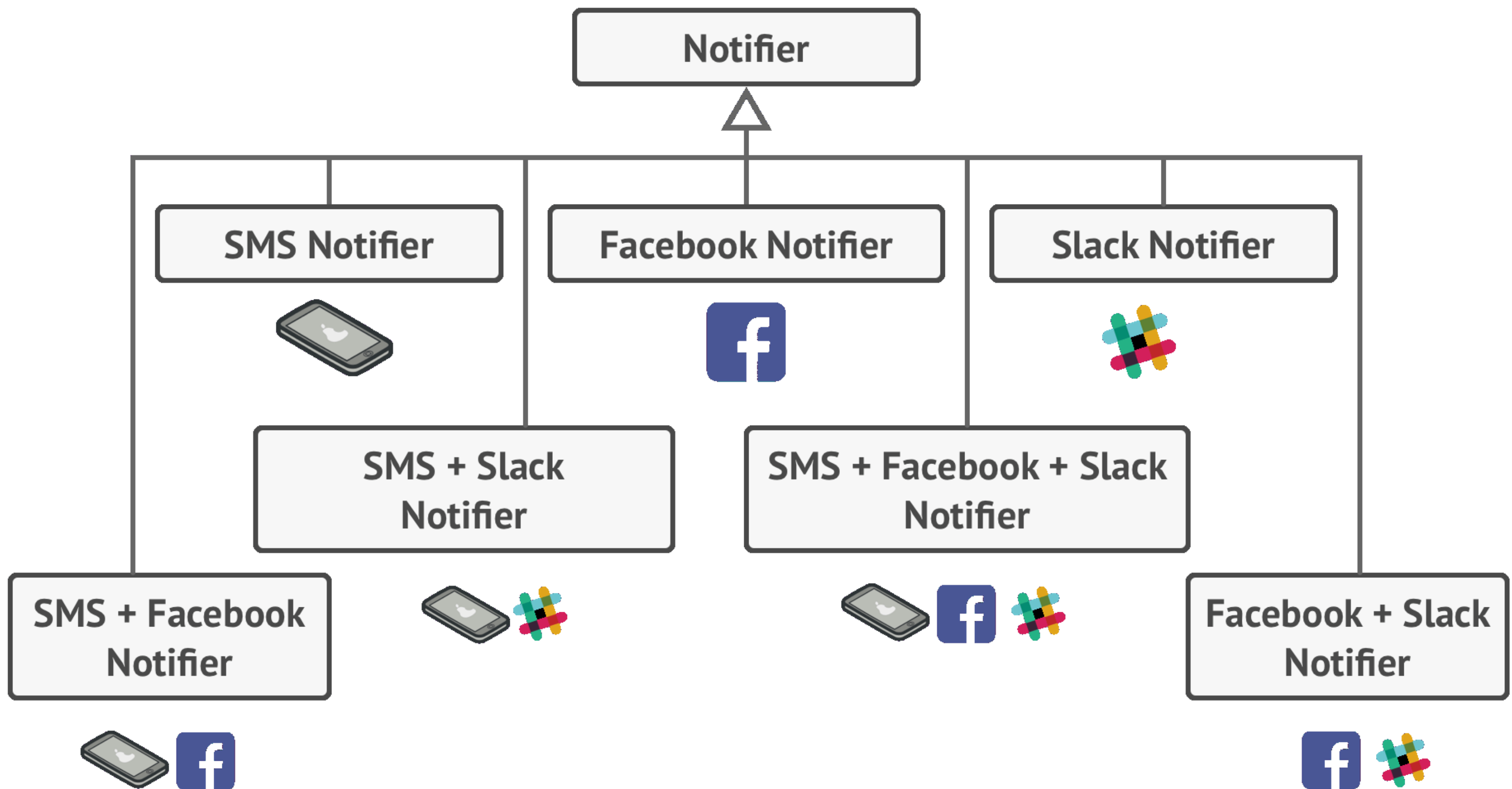
A program could use the notifier class to send notifications about important events to a predefined set of emails.

Contd.



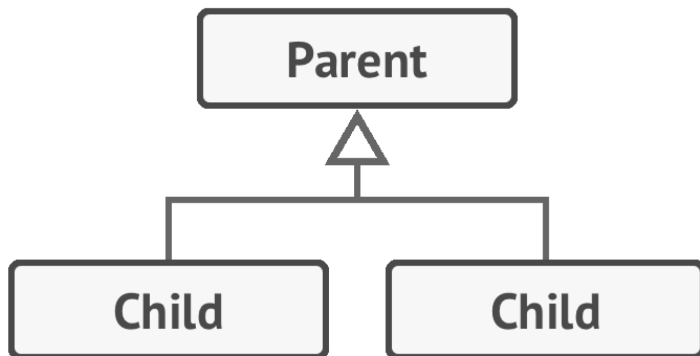
Each notification type is implemented as a notifier's subclass

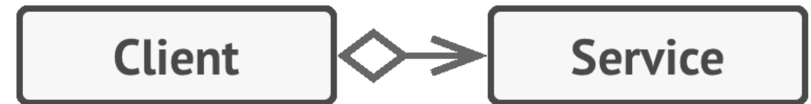
Contd.



Combinatorial explosion of subclasses.

Solution

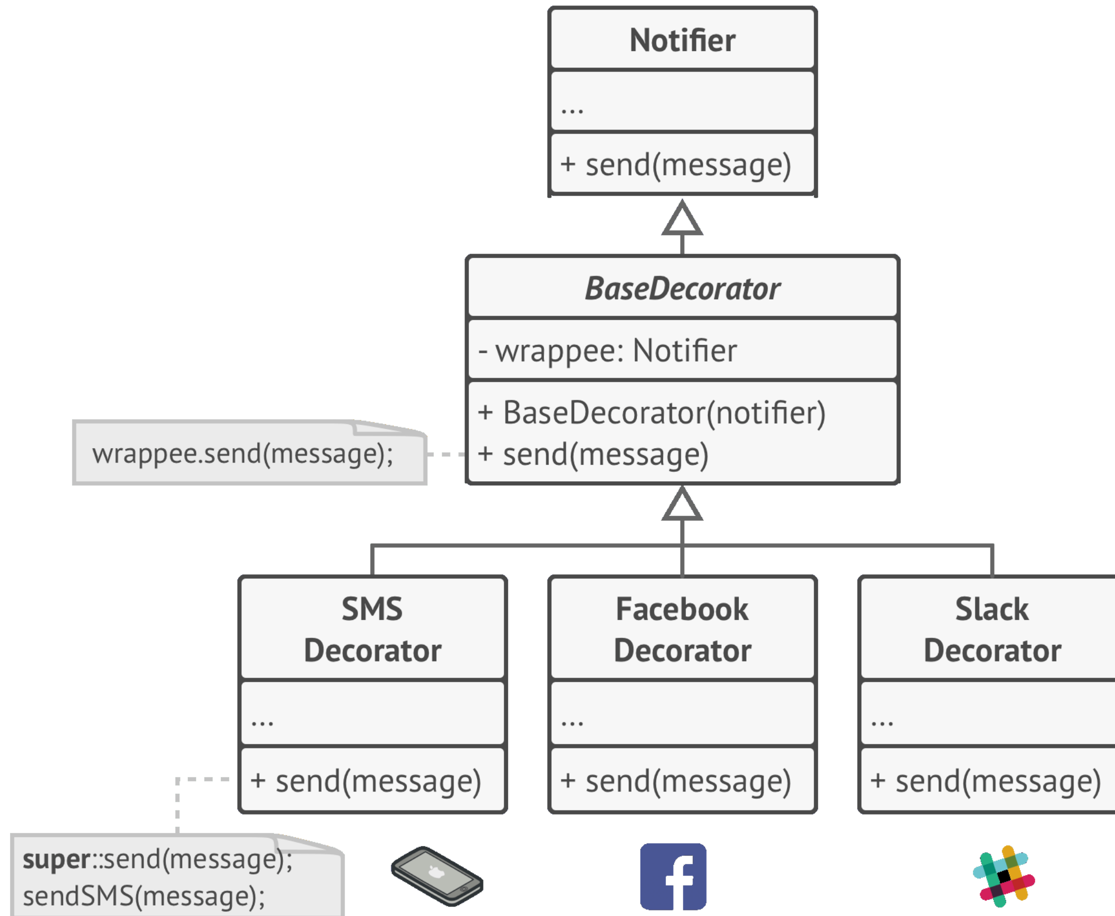




Composition

Inheritance vs. Composition

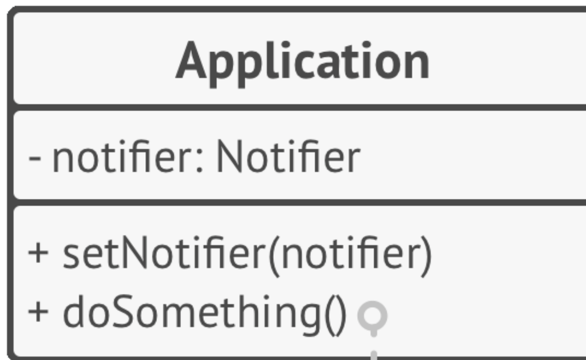
Contd.



Various notification methods become decorators

Contd.

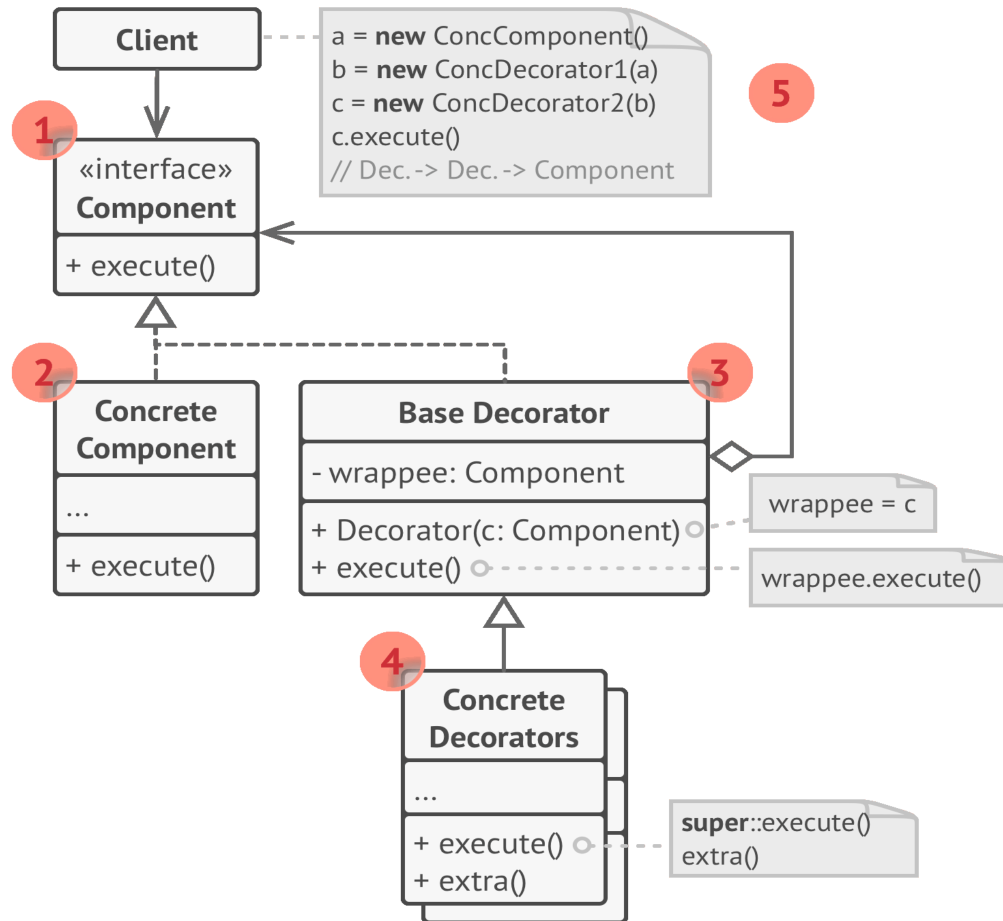
```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```



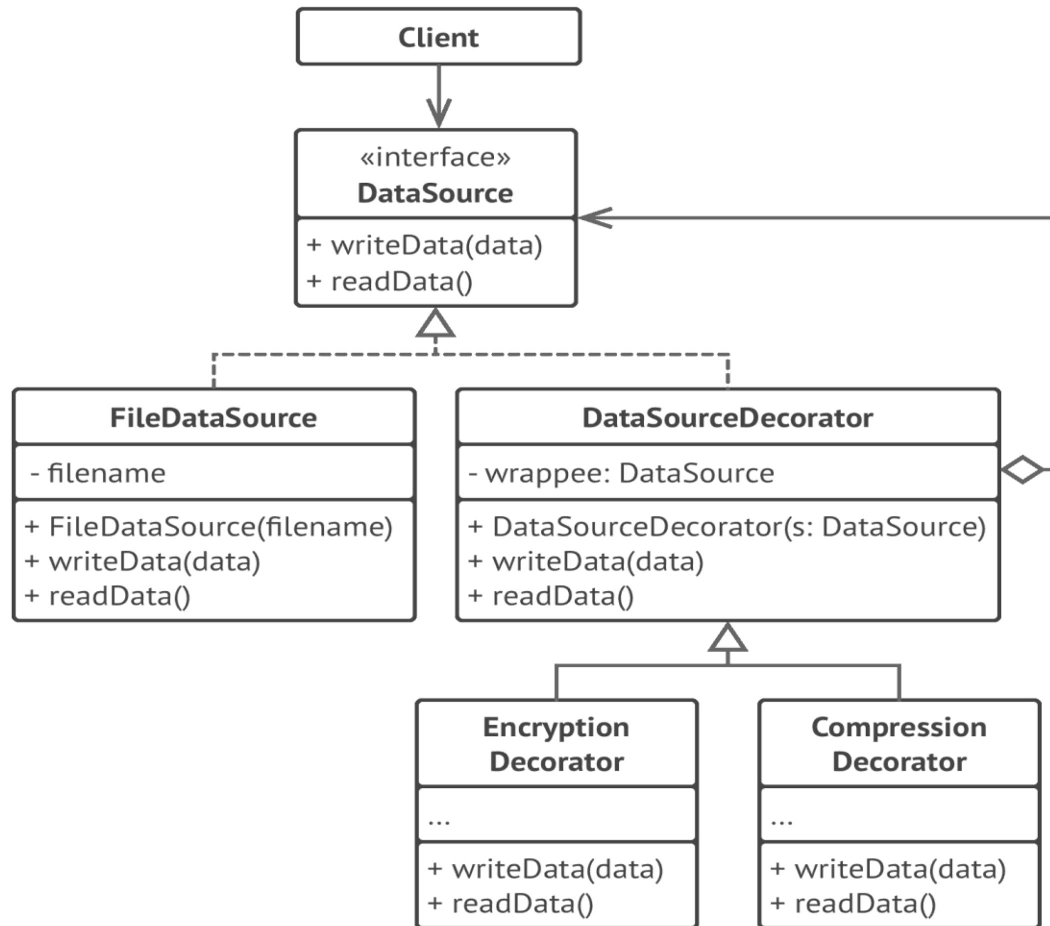
```
notifier.send("Alert!")
// Email → Facebook → Slack
```

Apps might configure complex stacks of notification decorators.

Structure



Implementation



The encryption and compression decorators example.

Application

- Decorator pattern is to be used when it is needed to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
- When it's awkward or not possible to extend an object's behavior using inheritance.

Pros & Cons

- It can be extended an object's behavior without making a new subclass.
- It can be added or removed responsibilities from an object at runtime.
- It can be combined several behaviors by wrapping an object into multiple decorators.
- *Single Responsibility Principle*. It can be divided a monolithic class that implements many possible variants of behavior into several smaller classes.
- It's hard to remove a specific wrapper from the wrappers stack.
- It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.