SE 350 - Object-Oriented Software Development

Software Design Patterns: Decorator

Instructor: Stefan Mitsch

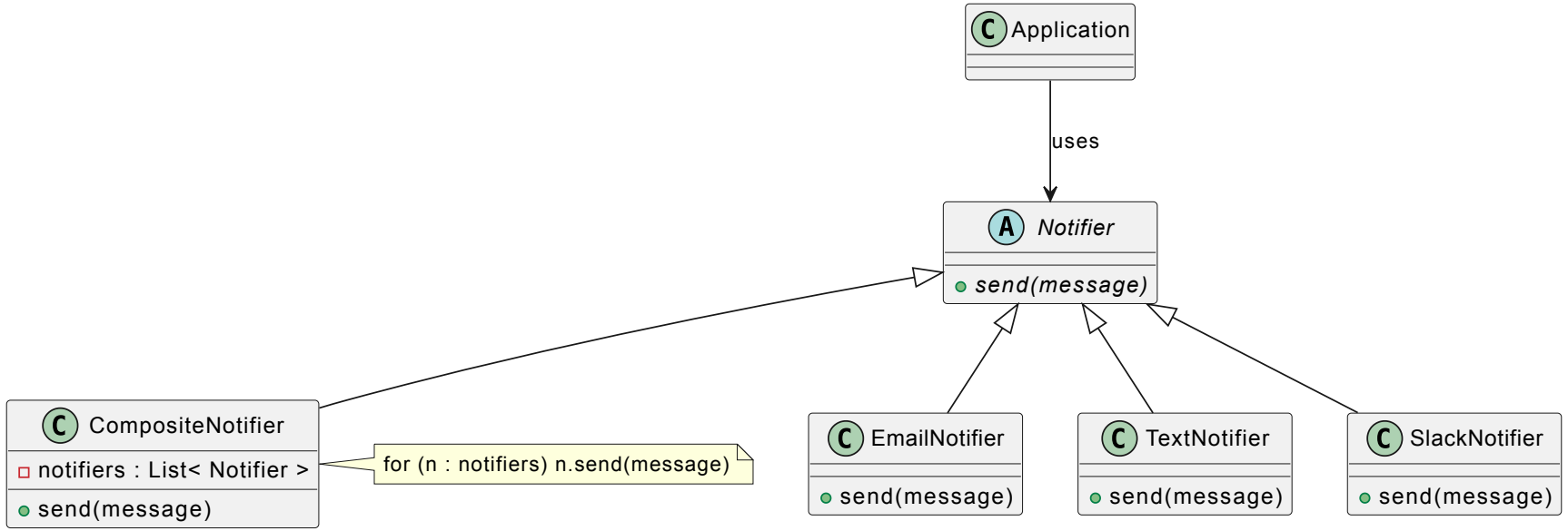
Learning Objectives

- Understand the problem solved by Decorator
- Understand the pattern

Design Challenge

Requirements

- application that can send notifications by email, text message, Slack
- users should be able to sign up for any combination of notifications

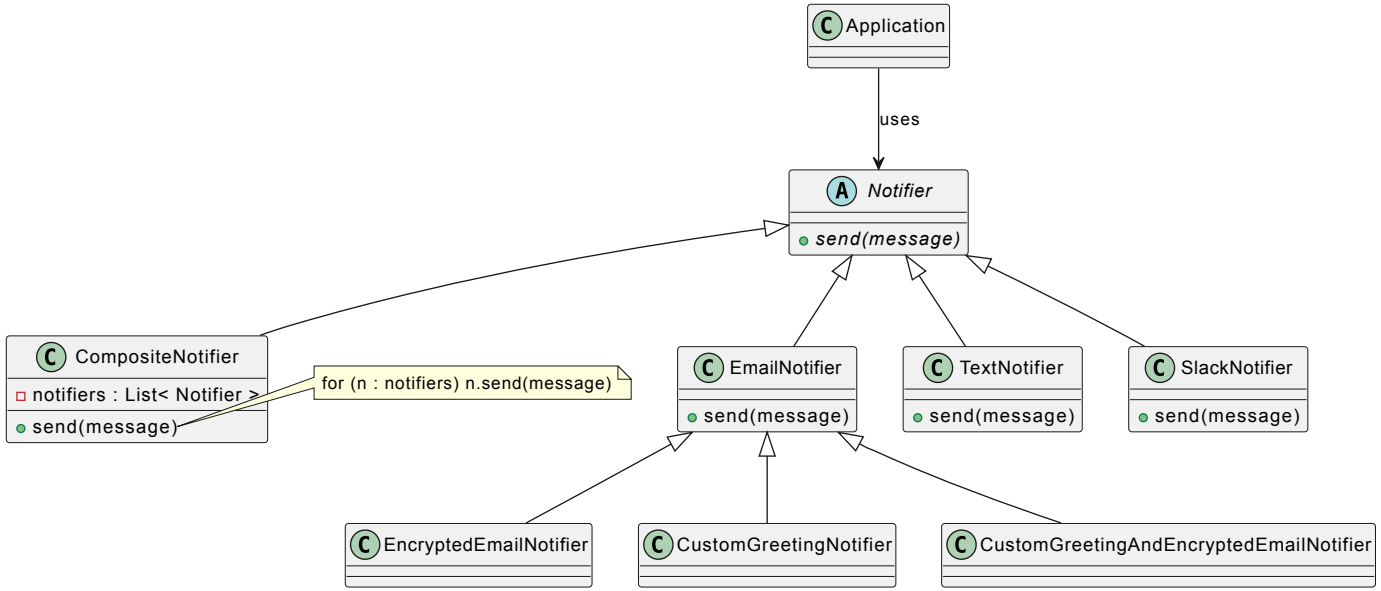


- users should be able to customize the `EmailNotifier` with a custom greeting, and enable or disable encryption

Design Challenge

Requirements

- application that can send notifications by email, text message, Slack
- users should be able to sign up for any combination of notifications
- users should be able to customize the `EmailNotifier` with a custom greeting, and enable or disable encryption



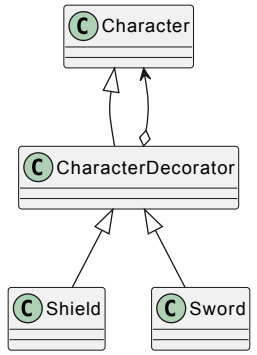
Decorator Introduction

Decorator adds functionality to an object dynamically

- Structural design pattern
- Dynamic alternative to subclassing
- Layering objects on top of each other

Problem illustration: want to add and remove functionality at runtime.

❓ How to extend an object without subclassing?



Decorator Overview

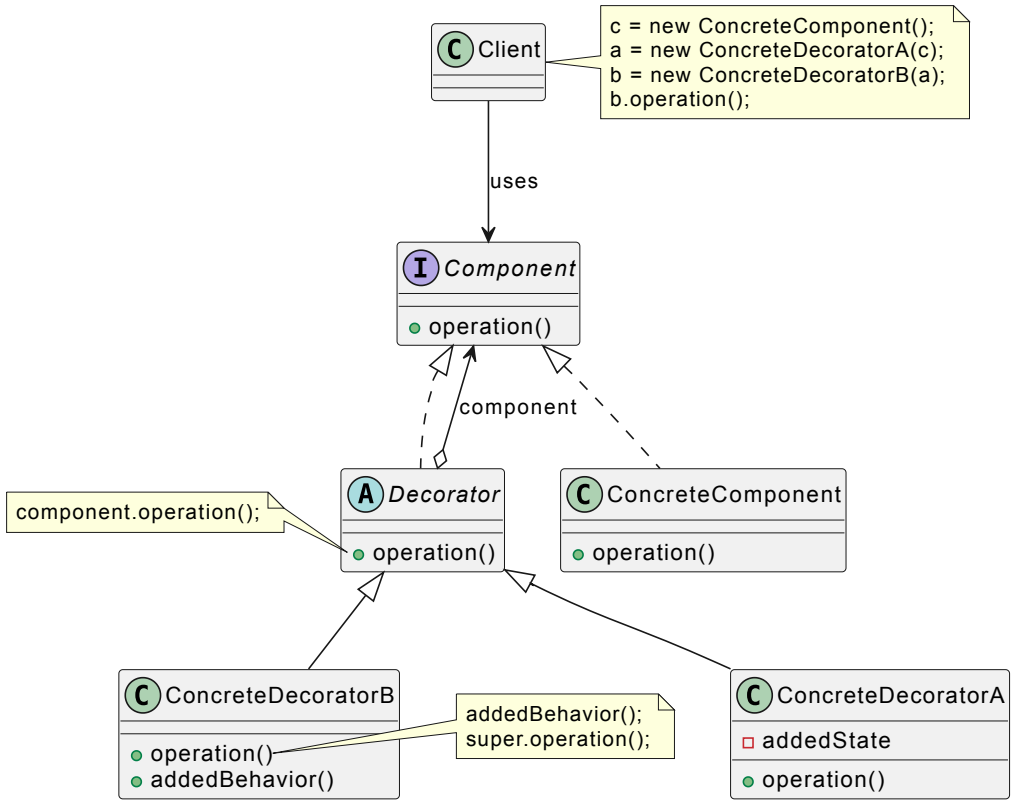
⚠ Problem

- Add behavior or state to object at runtime
- Add multiple layers of behavior

Intent

- Use to attach additional responsibilities to objects dynamically
- Use as an alternative to direct subclassing

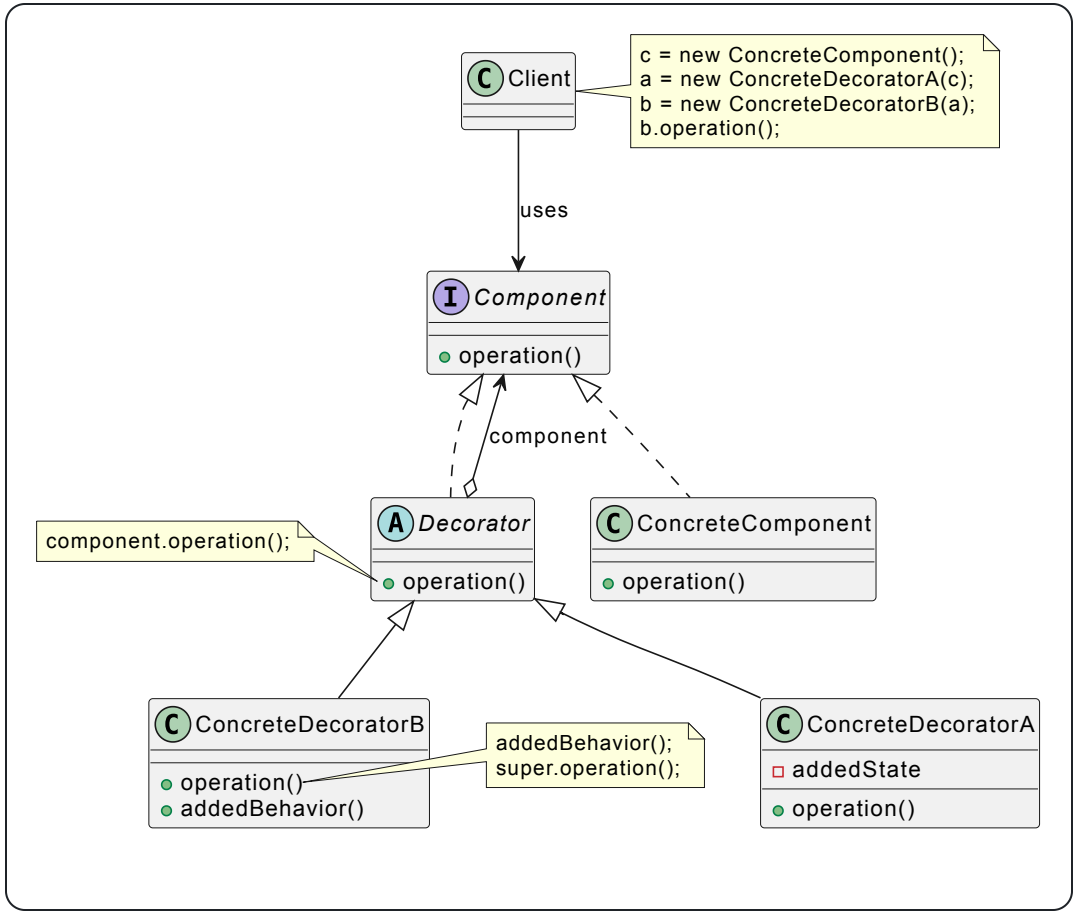
Structure



Decorator Participants

1. The `Component` interface declares operations that are common to both wrappers and wrapped objects

2. The `ConcreteComponent` is a class of objects being wrapped; it defines basic behavior, which can be extended by decorators.

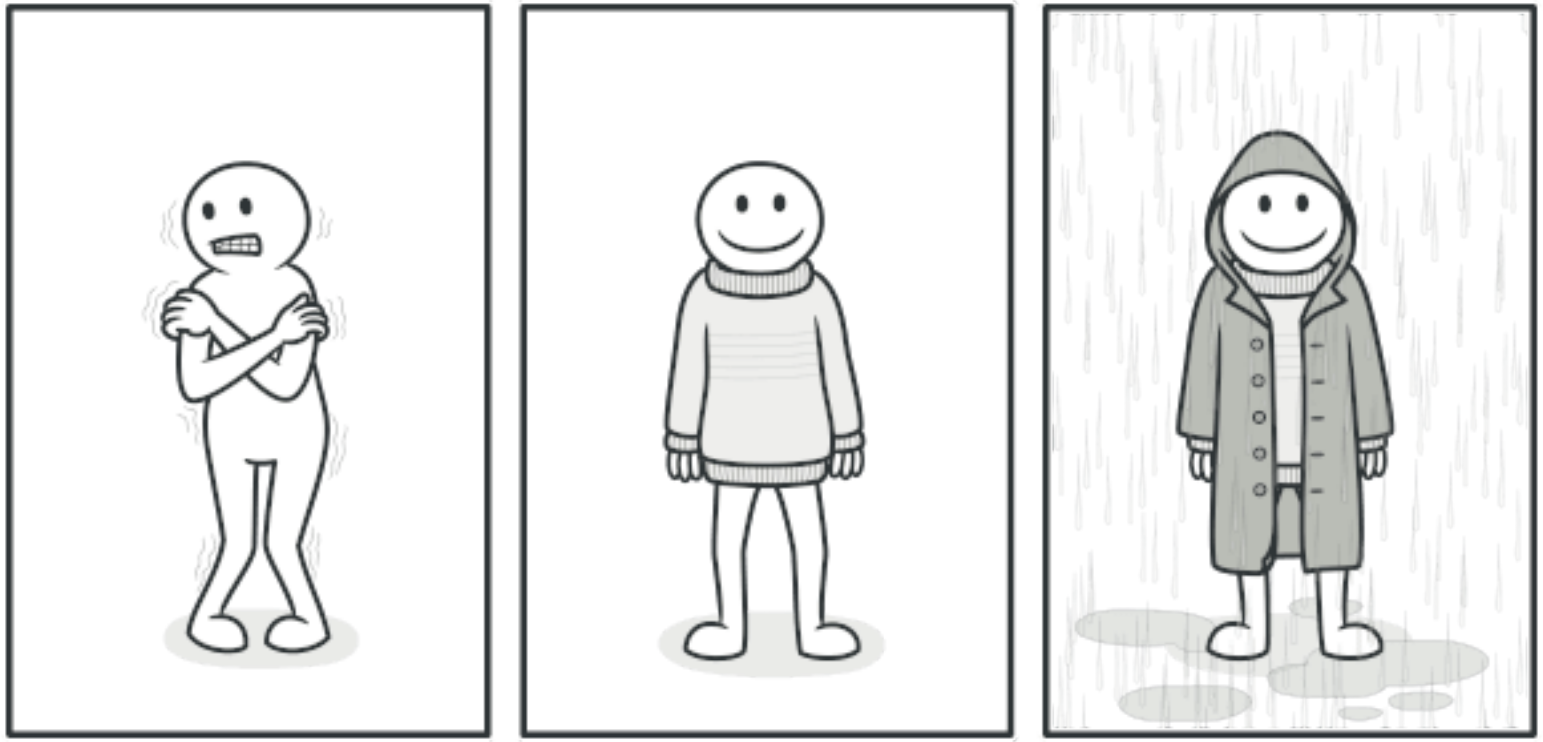


3. The `Decorator` class references the wrapped object

4. The `ConcreteDecorator` classes define extra behavior that can be added to the wrapped object dynamically; concrete decorators execute their additional behavior either before or after the base behavior

5. The `Client` sets up of layers of decorators

Decorator Real-World Analogy



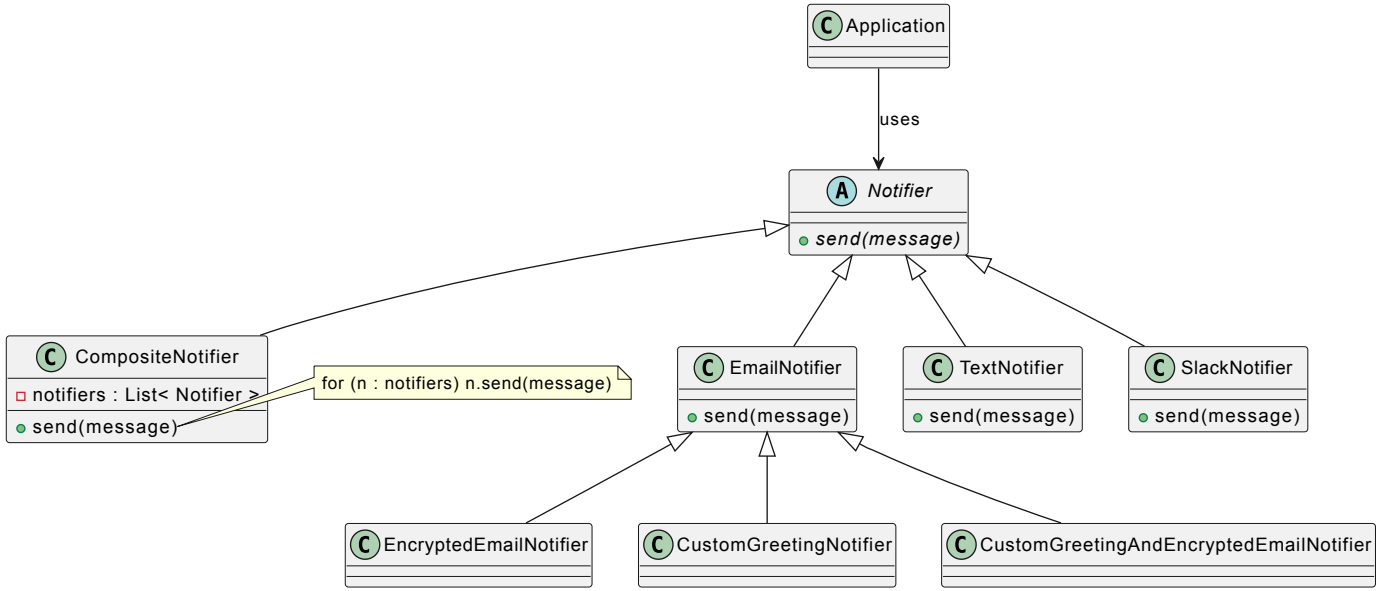
Add multiple layers of clothing to improve insulation

Image from "Dive into Design Patterns"

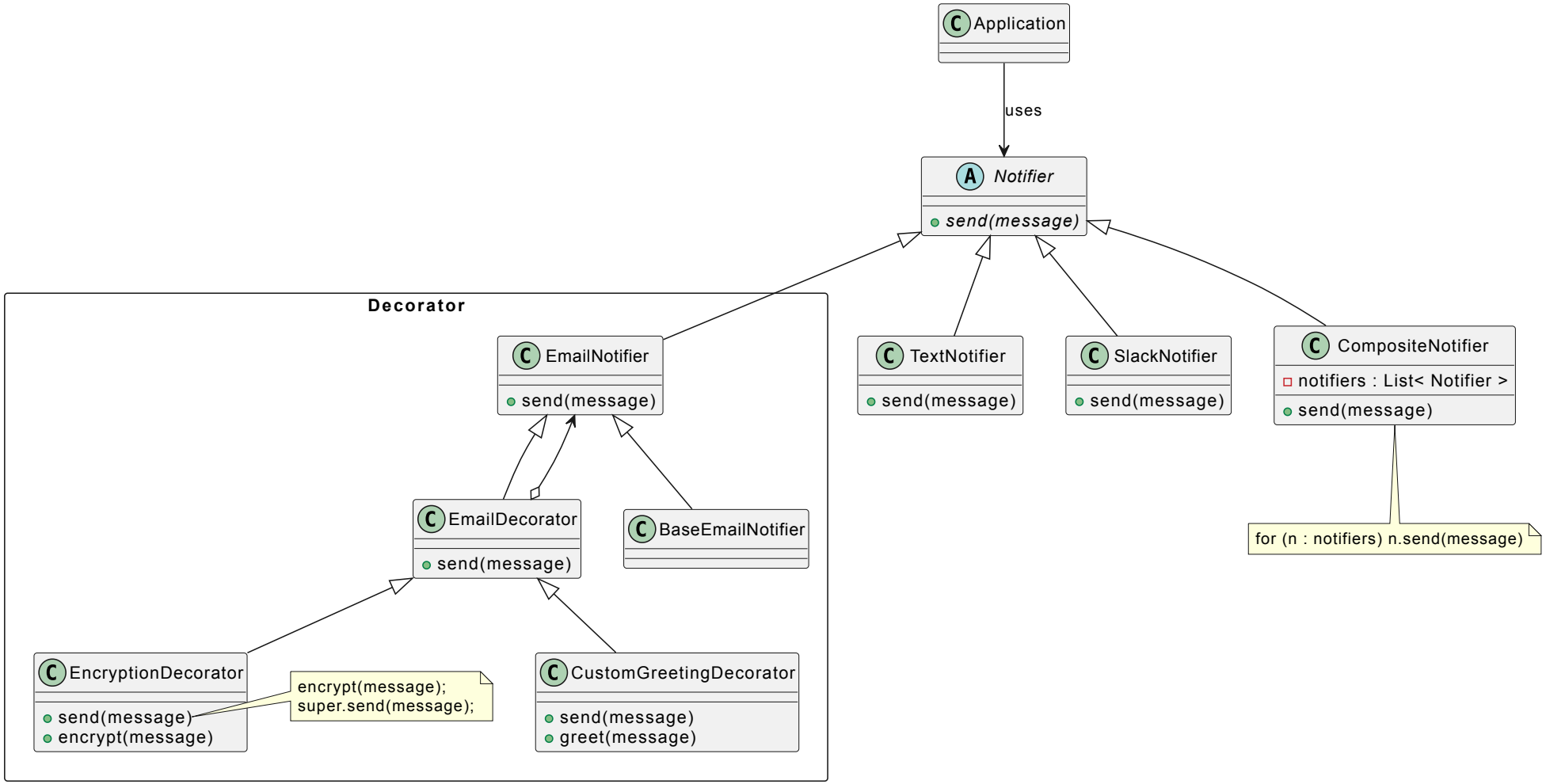
Design Challenge Revisited

Requirements

- application that can send notifications by email, text message, Slack
- users should be able to sign up for any combination of notifications
- users should be able to customize the `EmailNotifier` with a custom greeting, and enable or disable encryption



Design Challenge Improved



Implementation

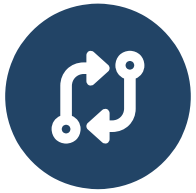
- Make sure the relevant parts of your app can be represented as **1 primary component** with multiple optional layers over it
- Define the `Component` interface as the common methods of primary object and layers
- Create a `ConcreteComponent` class
- Create a `Decorator` interface and implementing classes
- Compose decorators on top of the primary object in the client code

```
// common interface for object and decorators
public interface Character {
  void getStrength();
}

// primary object
public class BaseCharacter {
  private int strength;
}

// decorator class
public abstract class CharacterDecorator {
  private Character wrappee;
  public CharacterDecorator(Character wrappee) {
    this.wrappee = wrappee;
  }
  public int getStrength() { return wrappee.getStrength(); }
}

// concrete decorators
public class Shield extends CharacterDecorator {
  private int strength;
  public int getStrength() { return strength + super.getStrength(); }
}
public class Sword extends CharacterDecorator { /*...*/ }
```



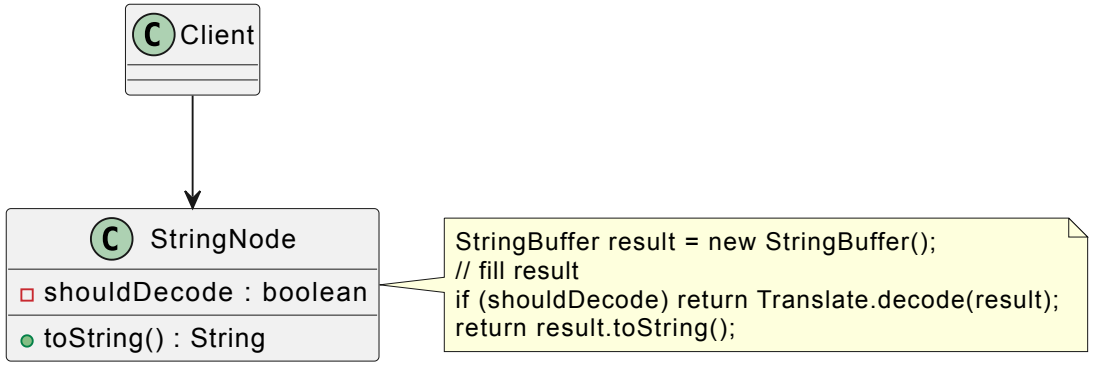
Decorator Variations

- Omit the `Decorator` interface when only 1 added behavior is needed
- Keep `Component` class light-weight: do not put too much code into the base component (ideally make it an `interface`)

Move Embellishment to Decorator

Code Smell: Code provides an embellishment to a class's core responsibility

Before: Conditional code



▶ After Refactoring

Discussion

Benefits

- Runtime modification of object state and behavior
- Combine multiple decorators to achieve combined effect
- Extend and remove functionality
- Promotes the Single Responsibility Principle: divide a monolothic class into multiple smaller decorators

Drawbacks

- Hard to remove an intermediate decorator
- Order of decorators may matter (e.g., add greeting, then encrypt)
- Setting up the decorators is not pretty code

Relations with Other Patterns

- **Composite** vs **Decorator**
  - Have very similar class diagrams
  - `Decorator` adds behavior, `Composite` sums up results
- `Decorator` changes the "skin" of an object; if you want to change its "guts", use `Strategy`