SE 350 - Object-Oriented Software Development

Software Design Patterns: Strategy

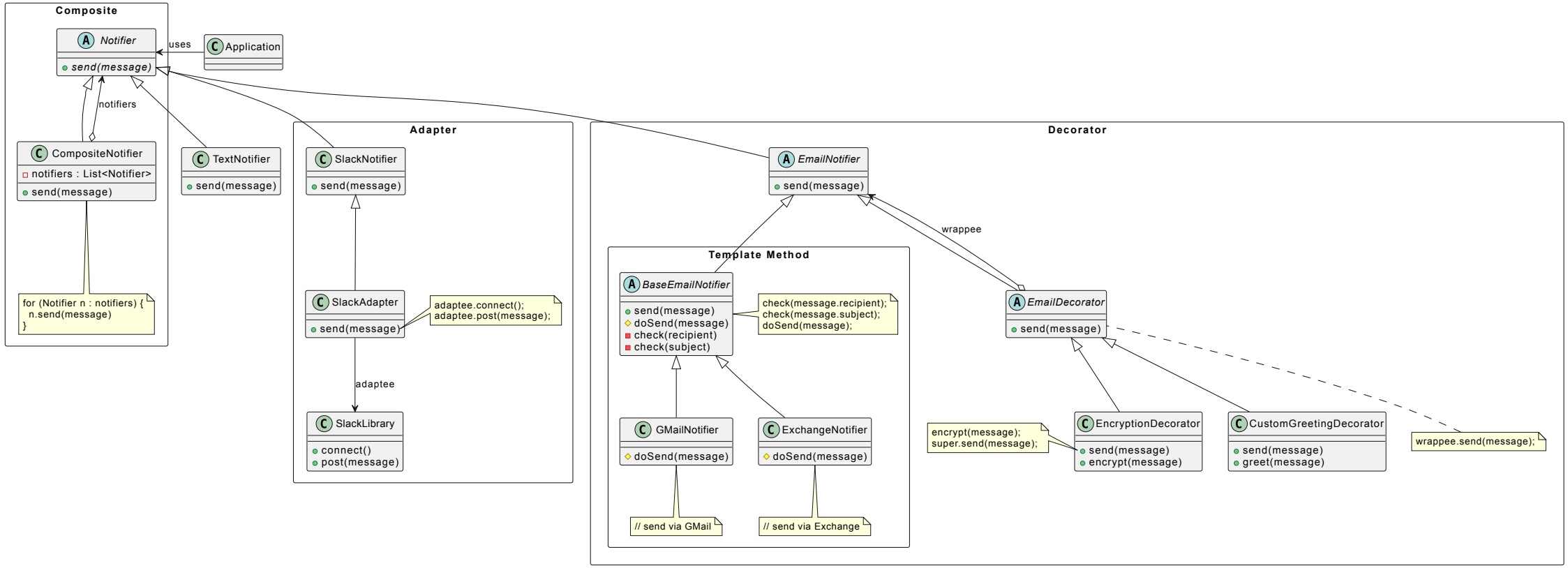
Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Strategy
- Understand the pattern

Design Challenge

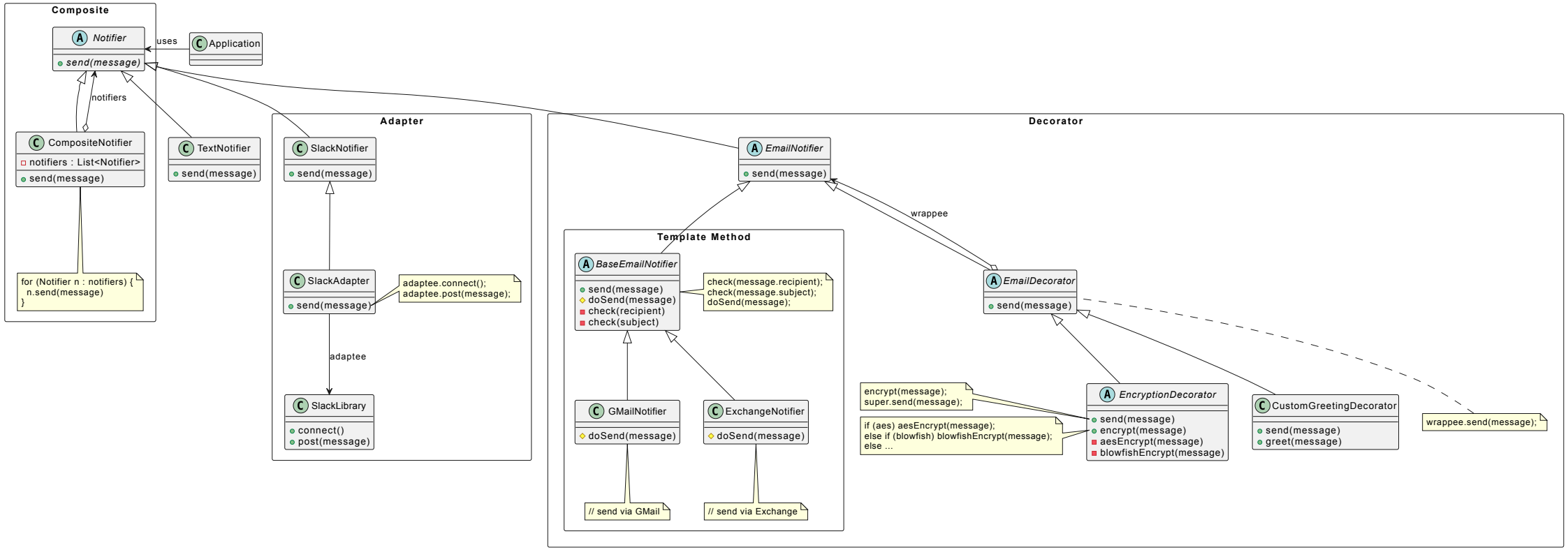
✓ Application that can send notifications by email, text message, Slack



? Adapt to user-preferred encryption (switch out encryption algorithm at runtime)

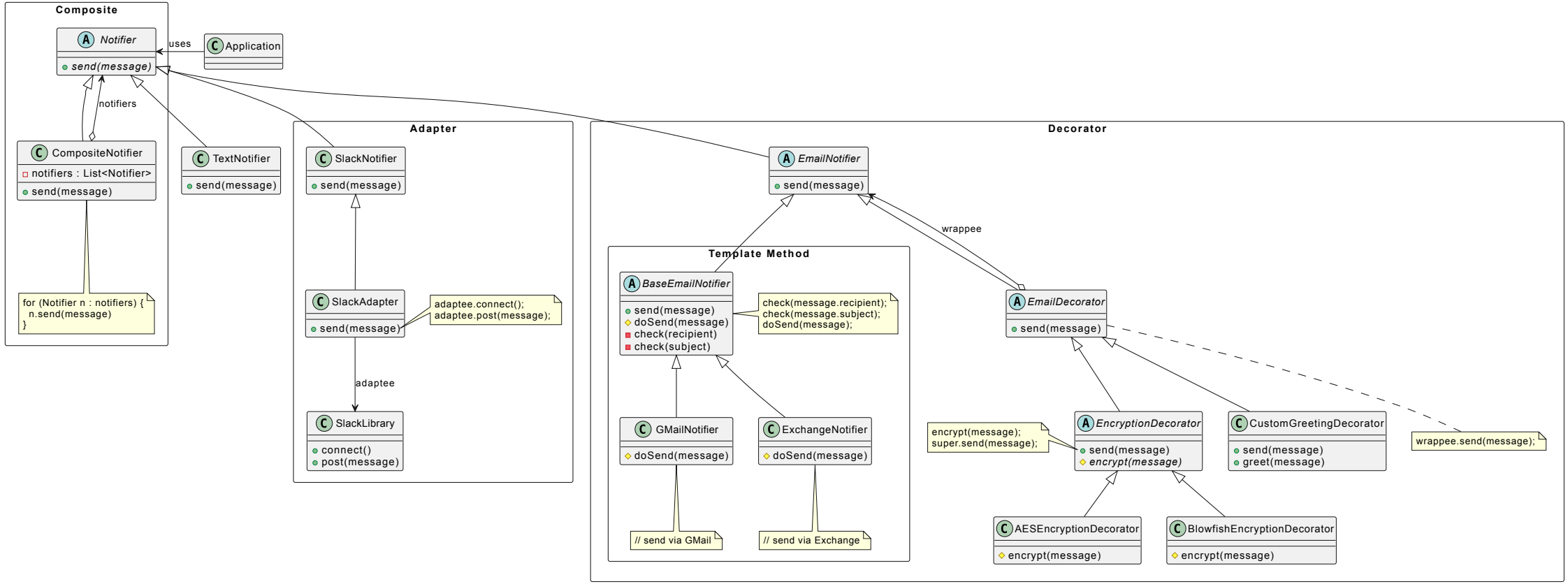
Design Challenge

- ✓ Application that can send notifications by email, text message, Slack
- ✓ Multiple encryption methods
- ✗ Bloated code, violates Open/Closed principle



Design Challenge

- ✓ Application that can send notifications by email, text message, Slack
- ✓ Multiple encryption methods
- ✗ Better, but requires rewiring all decorator layers when switching out encryption



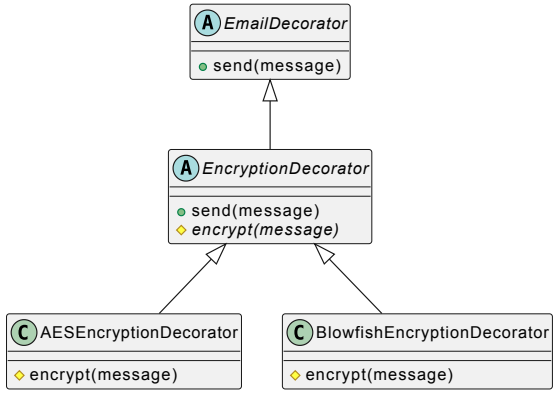
Strategy Introduction

Strategy defines a family of algorithms and makes them interchangeable at runtime

- Behavioral design pattern
- Lets clients switch out implementations

Problem illustration: switching out algorithm needs recompilation

❓ How to let application vary the encryption algorithm at runtime?



Strategy Overview

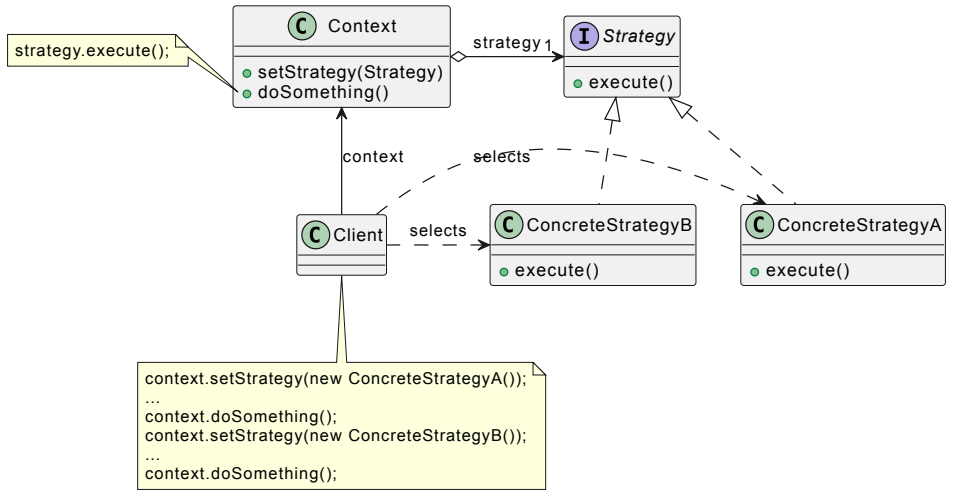
⚠ Problem

Several interchangeable algorithm implementations

Intent

- Use when many related classes differ only in behavior
- Need different variants of an algorithm
- Use to avoid conditional statements for switching out behavior

Structure

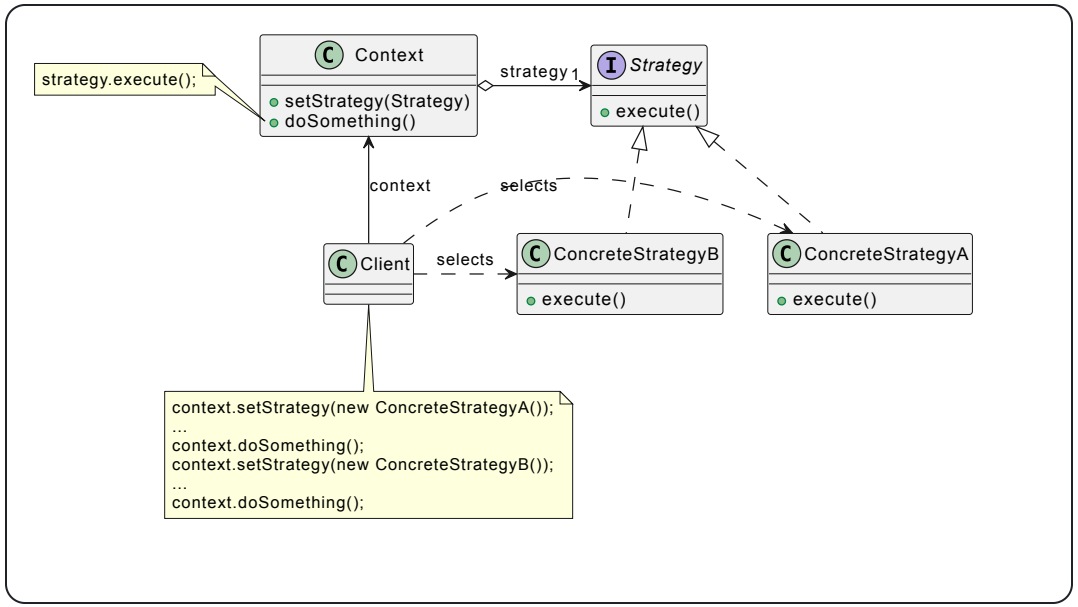


Strategy Participants

1. The `Context` maintains a reference to one of the concrete strategies

2. The `Strategy` interface declares methods for the `Context` to execute

3. `ConcreteStrategy` classes implement different variants of the algorithm

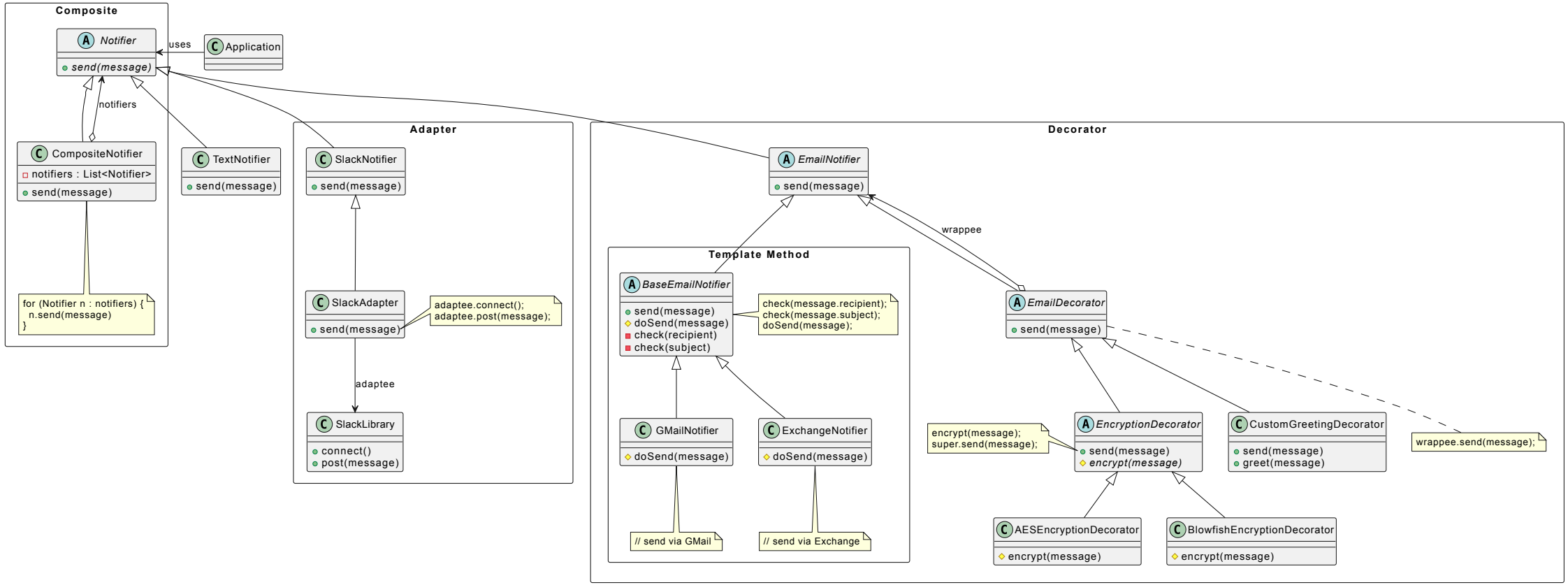


4. The context calls the `execute` method on the linked strategy each time it needs to run the algorithm; the context does not know which one it uses.

5. The `Client` creates a specific strategy object and passes it to the context; it then uses the context to execute the strategy

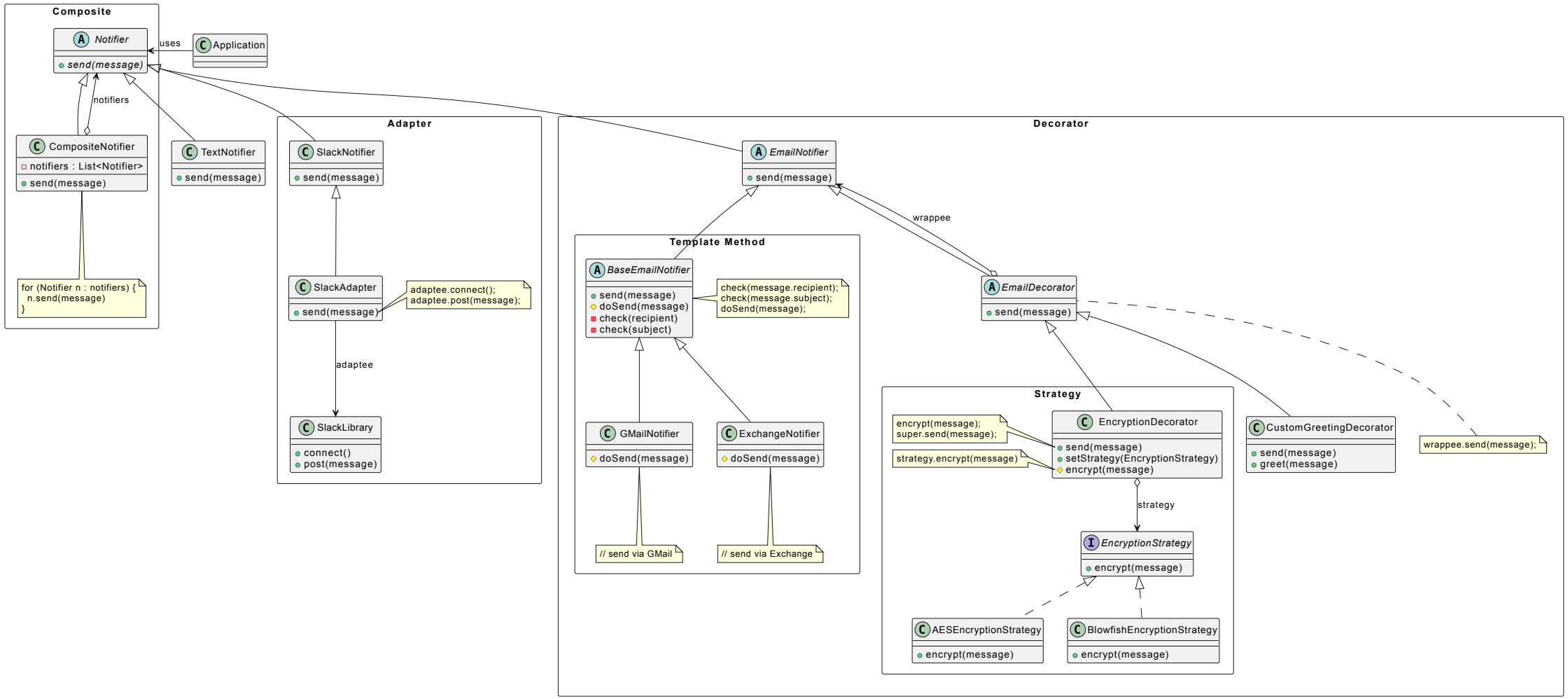
Design Challenge Revisited

- ✕ Requires rewiring all decorator layers when switching out encryption



Design Challenge Improved

- ✓ Switchable encryption strategy



Implementation

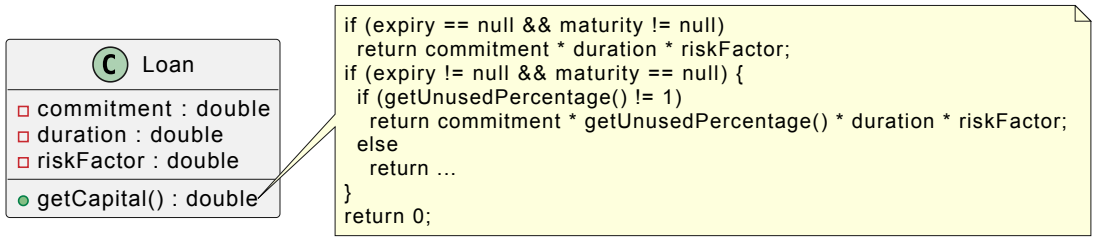
- In the context class, identify an algorithm that is prone to frequent change, or conditional statements to switch algorithms at runtime
- Declare a `Strategy` interface common to all algorithm variants
- Extract each algorithm into its own `ConcreteStrategy` class
- Store a reference to a `Strategy` in the `Context`
- Clients of the `Context` must set a strategy

```
// strategy interface
public interface EncryptionStrategy {
  void encrypt(Message m);
}

// concrete strategies
public class AESEncryptionStrategy implements EncryptionStrategy {
  public void encrypt(Message m) { /* ... */ }
}
public class BlowfishEncryptionStrategy implements EncryptionStrategy {
  public void encrypt(Message m) { /* ... */ }
}

// context
public class EncryptionDecorator extends EmailDecorator {
  private EncryptionStrategy strategy;
  public final void send(Message m) {
    encrypt(m);
    super.send(m);
  }
  public void setStrategy(EncryptionStrategy s) {
    strategy = s;
  }
  private void encrypt(Message m) {
    strategy.encrypt(m);
  }
}
```

Replace Conditional Logic with Strategy

Code Smell: Conditional logic in a method controls which fo several variants of a calcuation are executed

Before: Conditional logic for variants, frequently changes



▶ After Refactoring

Strategy Discussion

Benefits

- Can **swap algorithms at runtime**
- Promotes Single Responsibility Principle: can **isolate implementation details and variants** in own classes
- Replaces inheritance with composition
- Promotes Open/Closed Principle: can **add new algorithm variants**

Drawbacks

- Most beneficial **when many variants**
- Clients must be able to **select the proper concrete strategy** (must be aware of the differences)
- Support for **higher-order functions** enables similar behavior with less additional interfaces/classes

Relations with Other Patterns

- `Template Method` is based on inheritance; `Strategy` is a pattern to alter behavior based on composition (can change behavior at runtime)
- `Command` and `Strategy` look similar but have different intent: `Command` converts an operation into an object so it can be passed around and stored; `Strategy` swaps algorithms