# SE 350 - Object-Oriented Software Development

## Software Design Patterns: Template Method

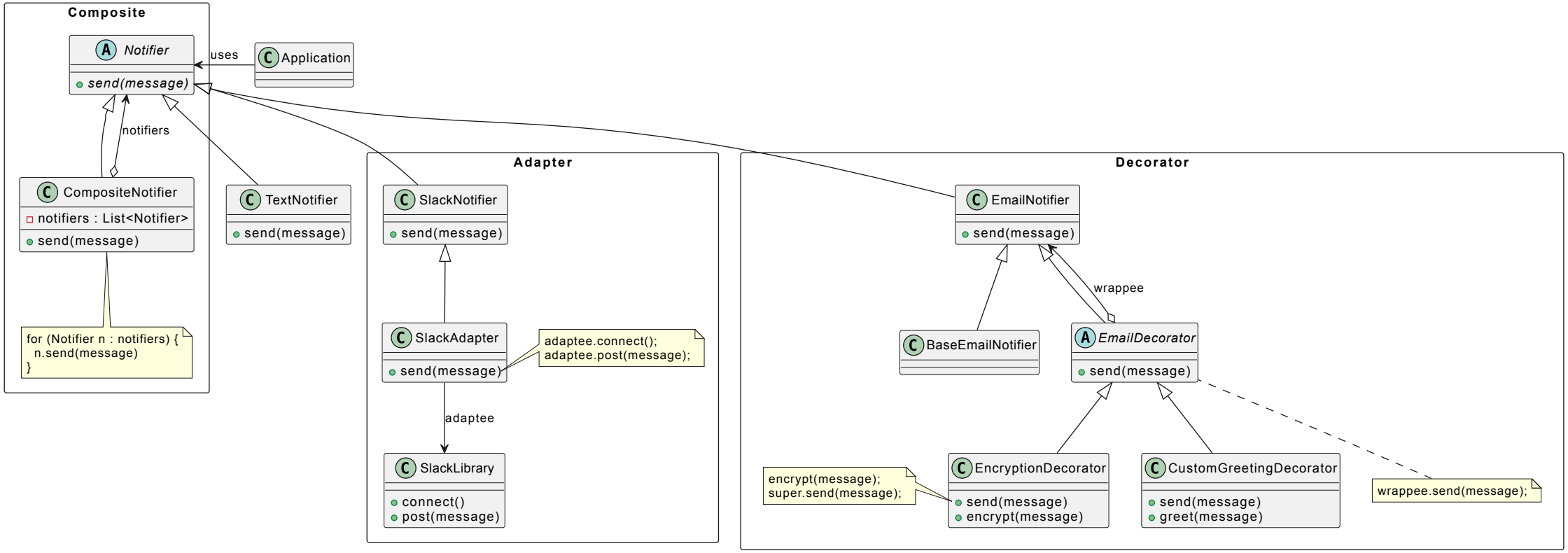
Instructor: Stefan Mitsch

# Learning Objectives

- Understand the problem solved by Template Method
- Understand the pattern

# Design Challenge

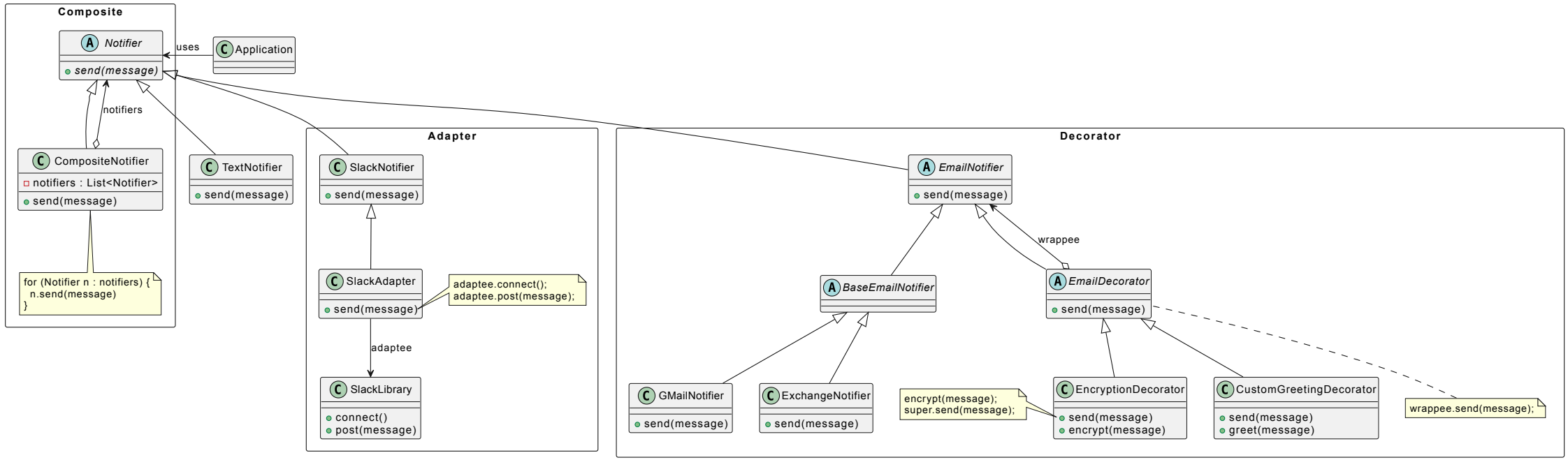
✓ Application that can send notifications by email, text message, Slack




? Adapt to our customer's email infrastructure, check message for empty recipient, check message for empty subject line

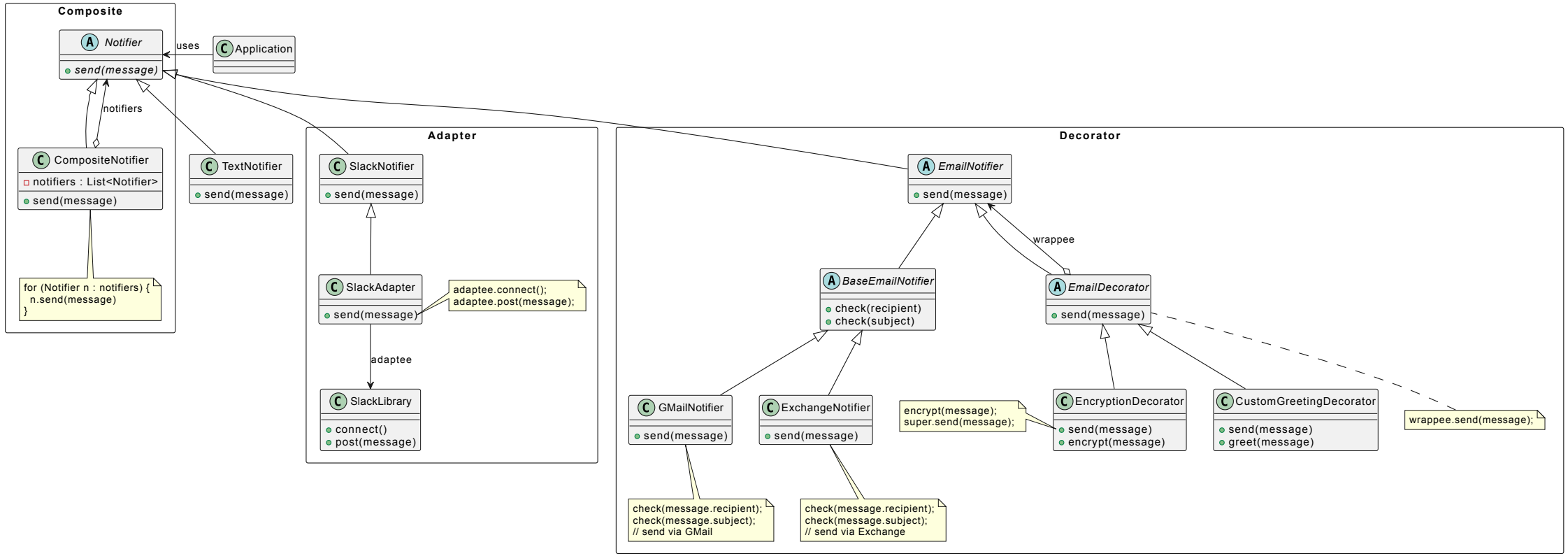
# Design Challenge

- ✓ Application that can send notifications by email, text message, Slack
- ✓ Adapt to our customer's email infrastructure
- ✗ Check message for empty recipient, check message for empty subject line



# Design Challenge

- ✓ Application that can send notifications by email, text message, Slack
- ✓ Adapt to our customer's email infrastructure, check message for empty recipient, check message for empty subject line




? What can be improved about this design?

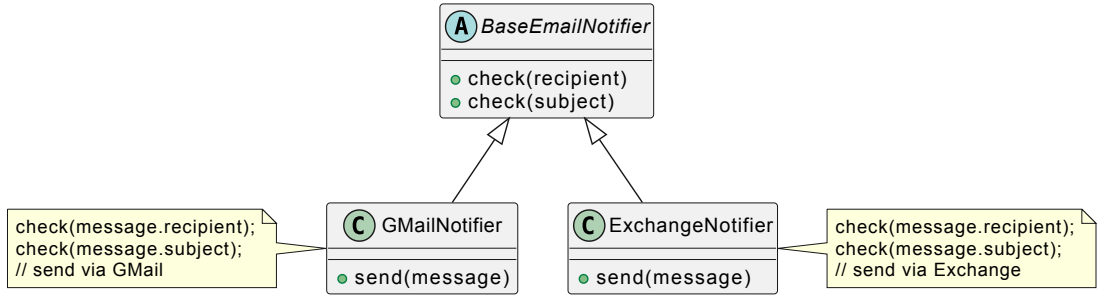
# Template Method Introduction

**Template Method defines the skeleton of an algorithm, deferring some steps to subclasses**

- Behavioral design pattern
- Lets subclasses redefine algorithm steps without changing the algorithm's structure

**Problem illustration:** algorithm structure duplicated over multiple classes

❓ How to define algorithm structure once for all subclasses, while letting them adapt some steps?



# Template Method Overview

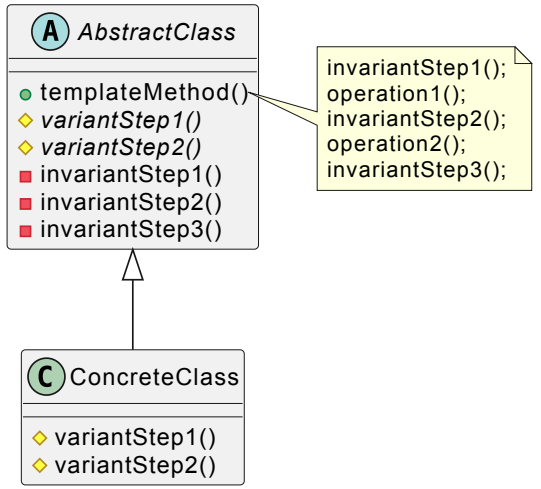
## Problem

Algorithm structure with some invariant steps and some changeable steps

## Intent

- Implement algorithm structure once, varying behavior in subclasses
- Factor common behavior into a base class to avoid code duplication
- Control how subclasses can extend an algorithm

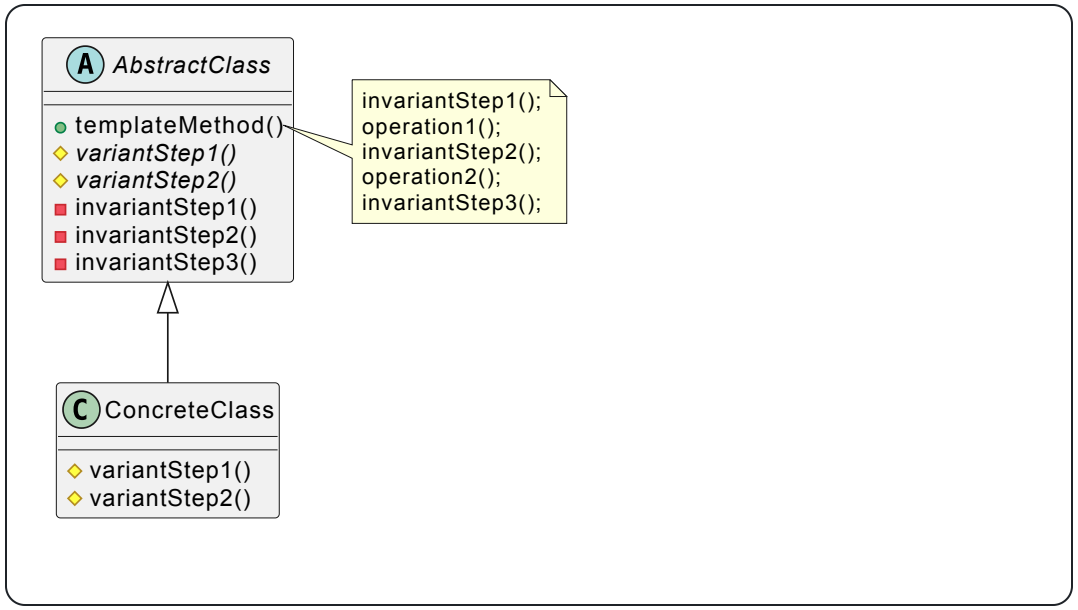
## Structure



# Template Method Participants

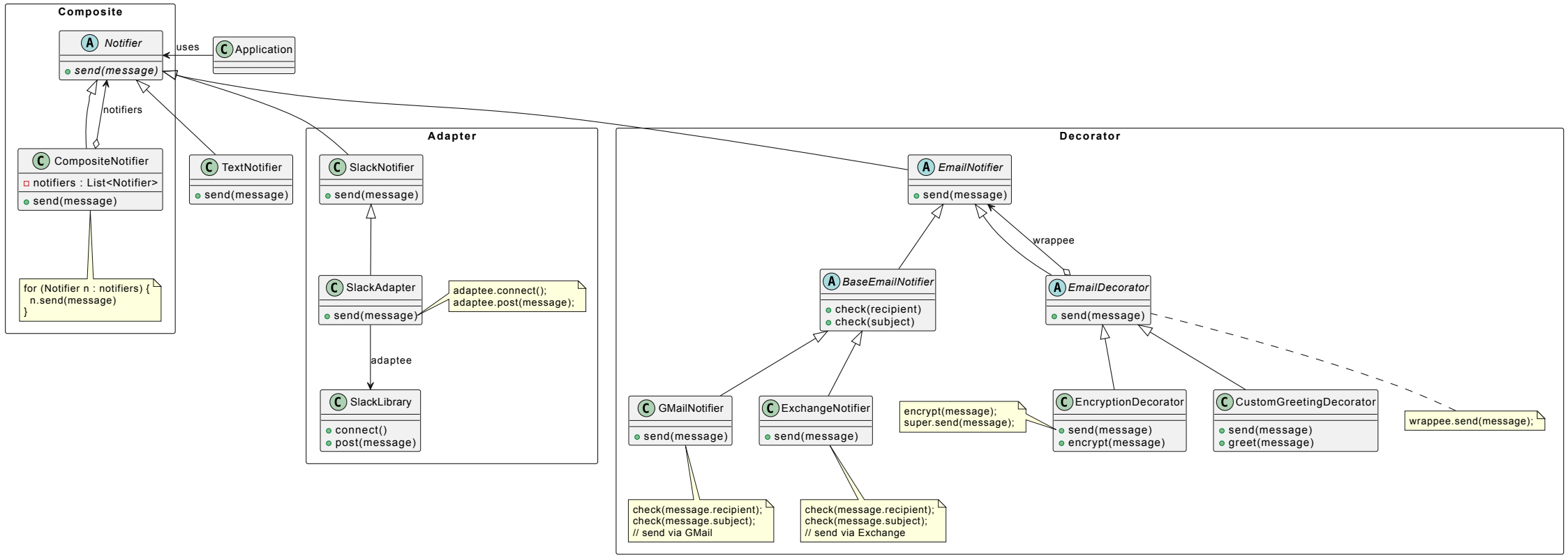
1. The `AbstractClass` defines concrete invariant steps and abstract variant steps

2. The `AbstractClass` defines a `templateMethod` that implements the algorithm structure by calling invariant and variant steps



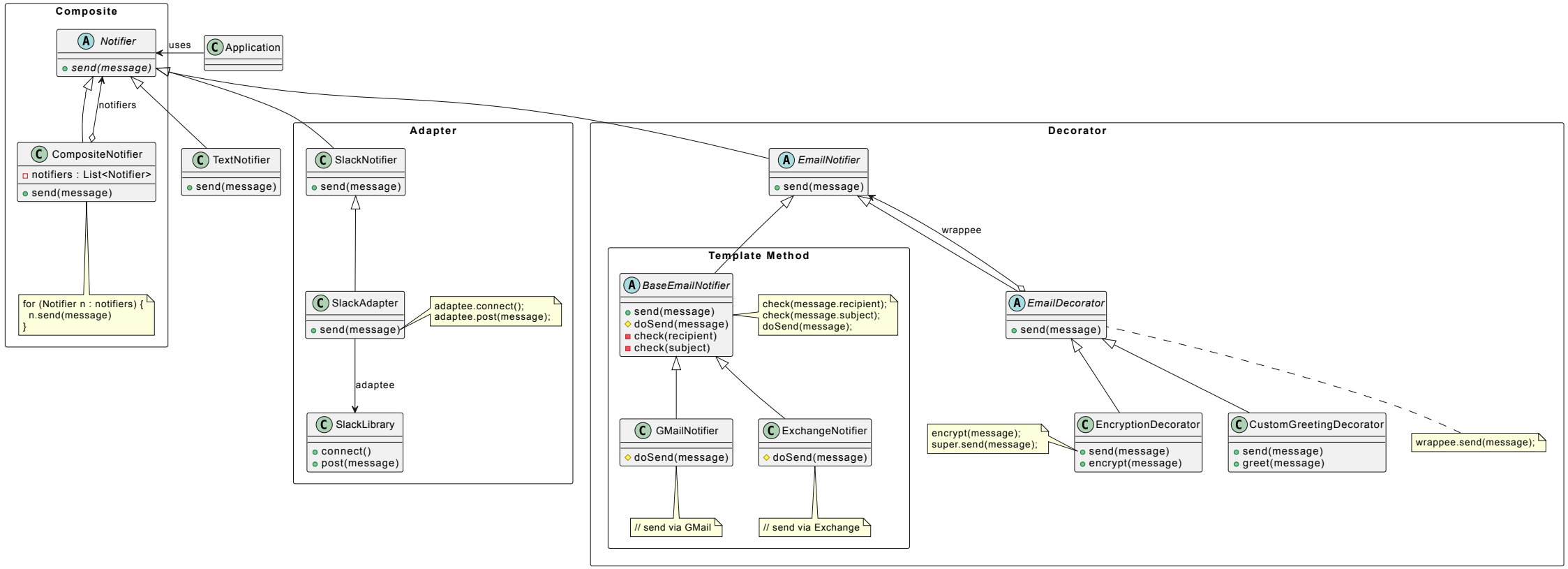

3. The `ConcreteClass` implements the subclass-specific variant steps

# Design Challenge Revisited




? What can be improved about this design?

# Design Challenge Improved



# Implementation

- Break the target algorithm into steps
- Create an abstract base class that declares the template method and a set of (abstract) steps
- Implement the algorithm sketch in the template method
- Provide default step implementations (it's ok if all steps are abstract)
- Consider making the template method `final`
- For each variation of the algorithm, implement abstract steps in a subclass

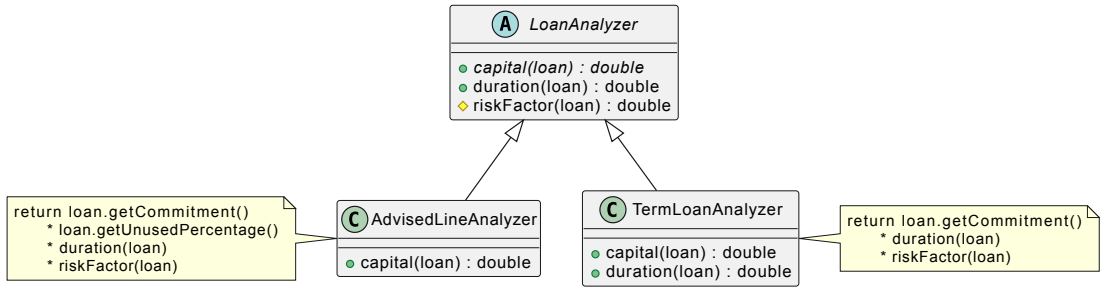
```
// base class with template method
public abstract class BaseEmailNotifier {
  // template method
  public final void send(Message m) {
    check(m.getRecipient());
    check(m.getSubject());
    doSend(m);
  }

  // steps mandatory to override
  protected abstract void doSend(Message m);

  // common steps
  private void check(Recipient r) { /* ... */ }
  private void check(Subject s) { /* ... */ }
}

// concrete algorithm variants
public class GMailNotifier extends BaseEmailNotifier {
  protected void doSend(Message m) { /* ... */ }
}
public class ExchangeNotifier extends BaseEmailNotifier {
  protected void doSend(Message m) { /* ... */ }
}
```

# Form Template Method

**Code Smell:** Two methods in subclasses perform similar steps in the same order, but some steps are different

Before: Algorithm duplicated in two subclasses

▶ After Refactoring



# Template Method Discussion

## Benefits

- Clients override only parts of a large algorithm
- Can pull duplicate code into a superclass
- Can enforce common algorithm steps

## Drawbacks

- Algorithm skeleton may be limiting to some implementations
- Can violate Liskov Substitution Principle when suppressing steps
- Harder to maintain when algorithm has many steps

# 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)