SE 350 - Object-Oriented Software Development

Software Design Patterns: Adapter

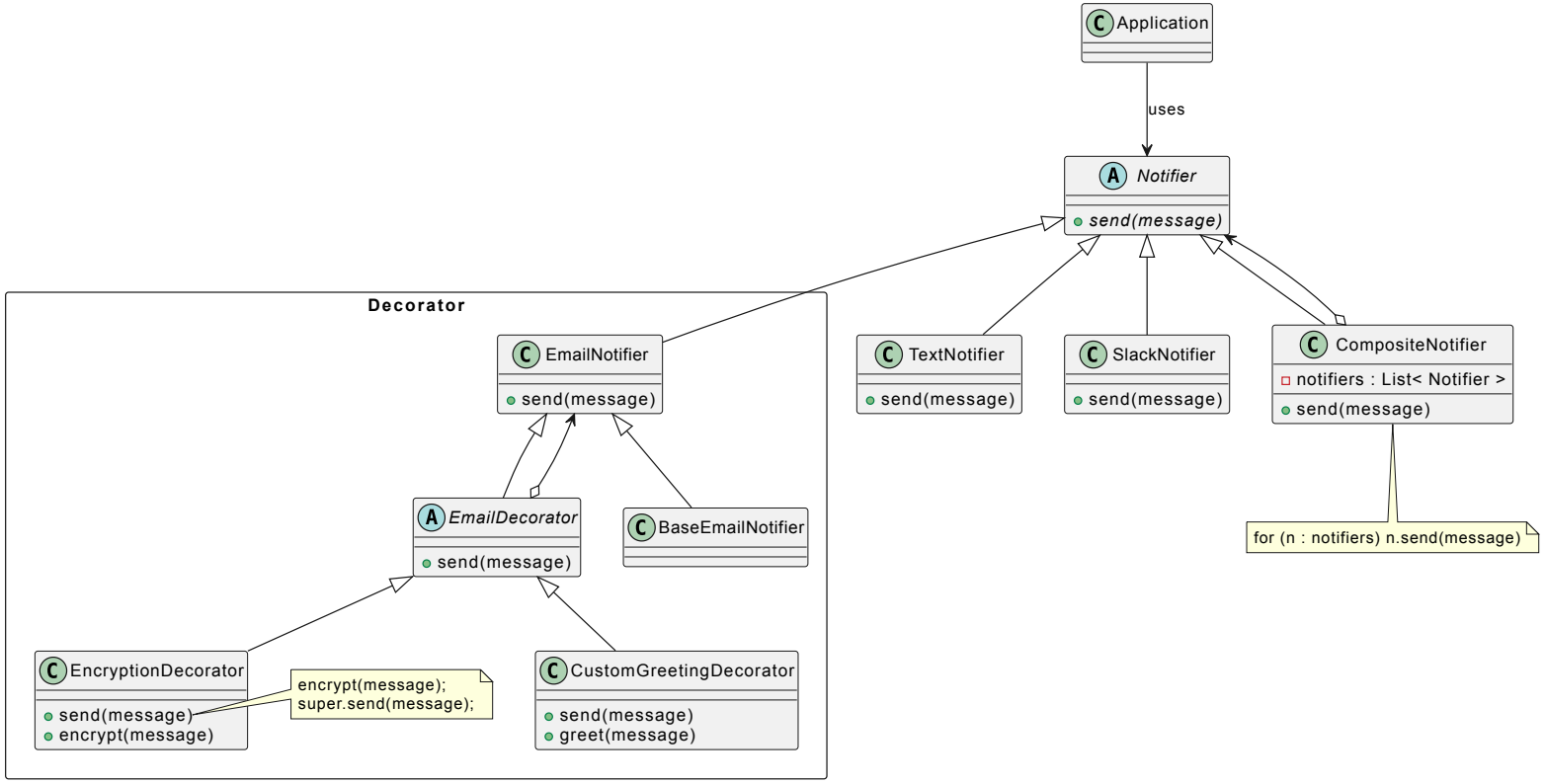
Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Adapter
- Understand the pattern

Design Challenge

- Application to send notifications over configurable channels



- We want to implement the `SlackNotifier` using a third-party library, whose interface does not fit our design

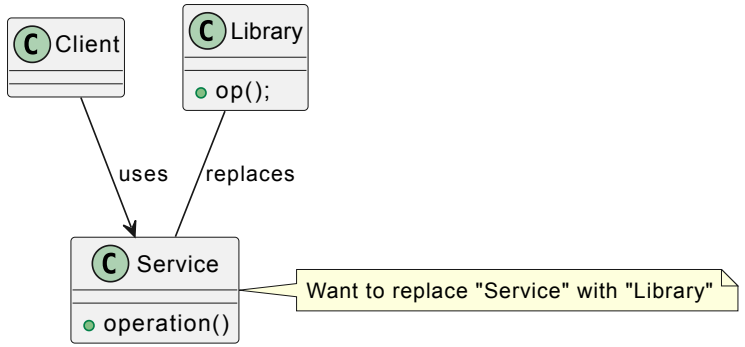
Adapter Introduction

Adapter converts an interface of a class into another interface

- Structural design pattern
- Commonly used with legacy code
- Commonly used when changing libraries

Problem illustration: want to use a different library with incompatible interface

❓ How to use library without changing existing client code?



Adapter Overview

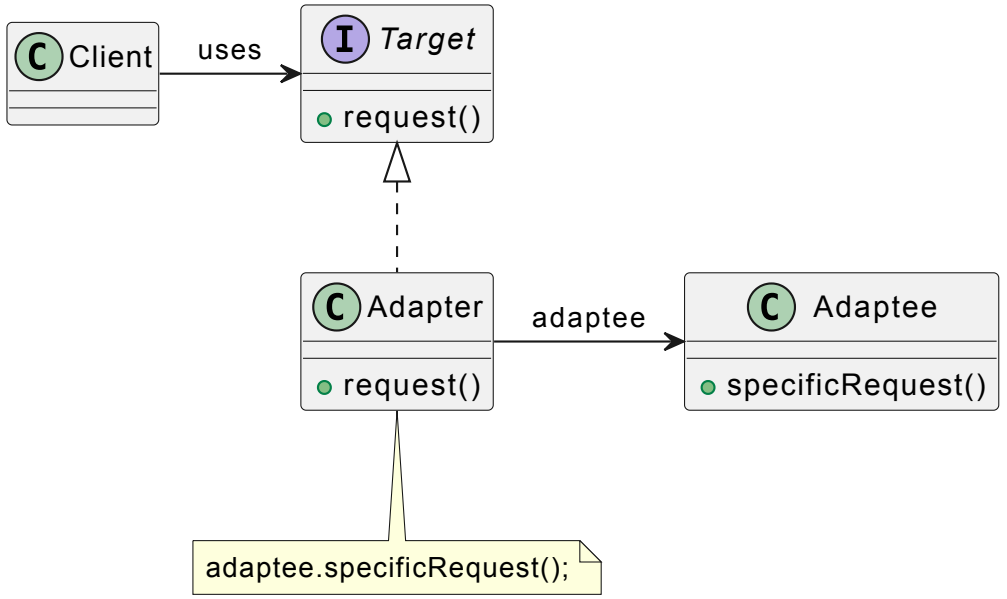
⚠ Problem

- Reuse legacy code with incompatible interface
- Change library with incompatible interface

Intent

- Adapt interfaces
- Wrap existing code

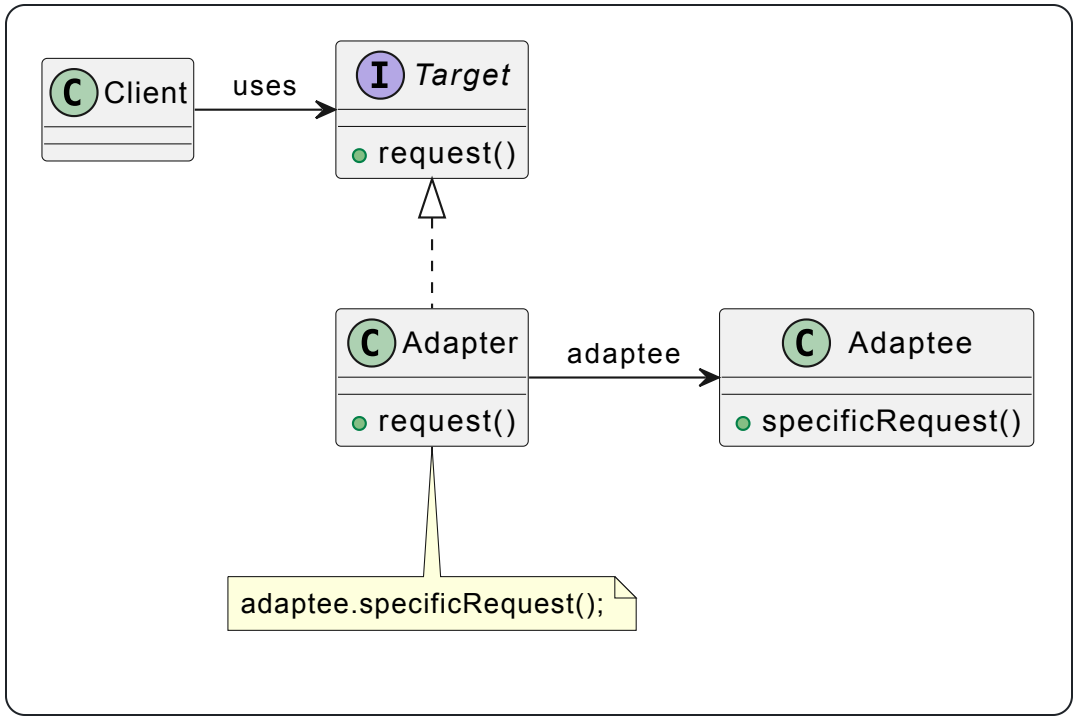
Structure



Adapter Participants

1. The `Client` contains the existing business logic of the program

2. The `Adaptee` provides a useful service (third-party or legacy) that doesn't work directly with the `Client` because of interface differences

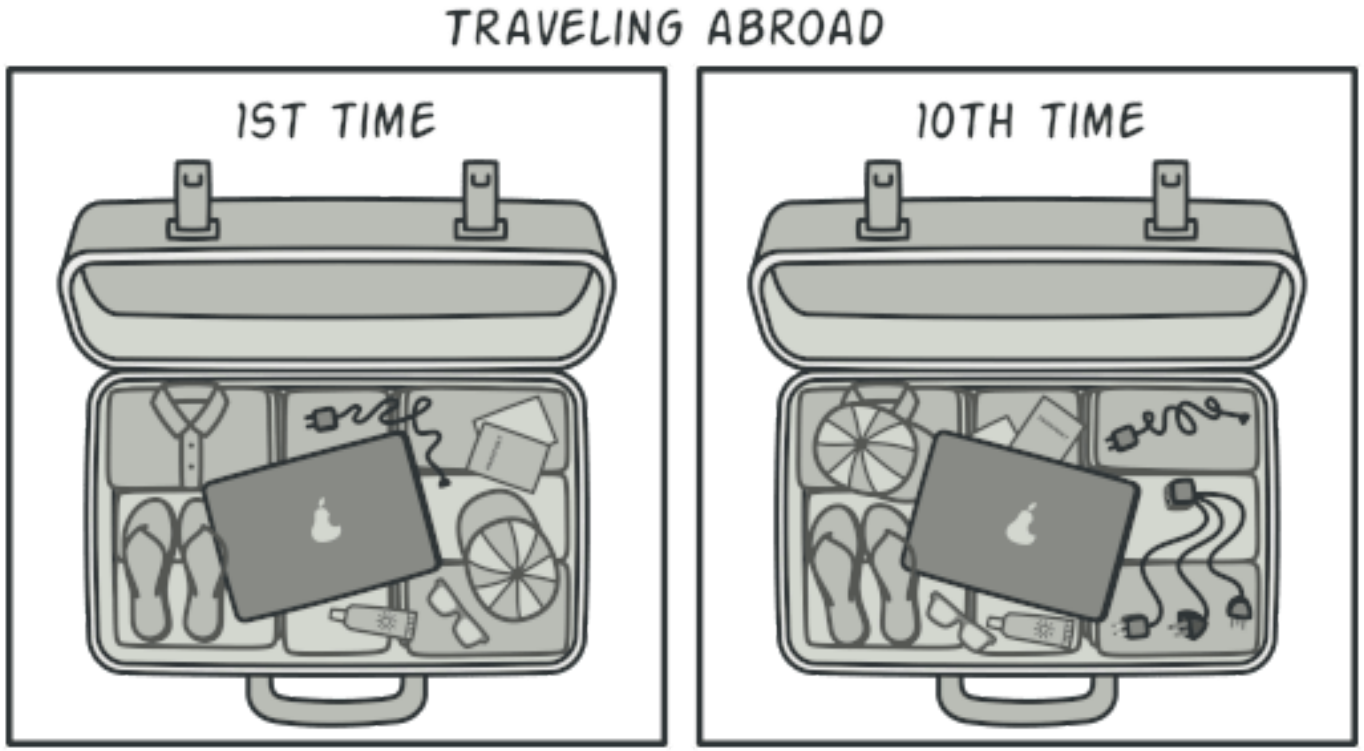


3. The `Target` interfaces defines the interface expected by the `Client`

4. The `Adapter` implements the `Target` interface by forwarding to the `Service`

5. The client code is decoupled from the concrete adaptee and from the adapter; thanks to the `Target` interface, new adapter implementations can be added without breaking the `Client`

Adapter Real-World Analogy

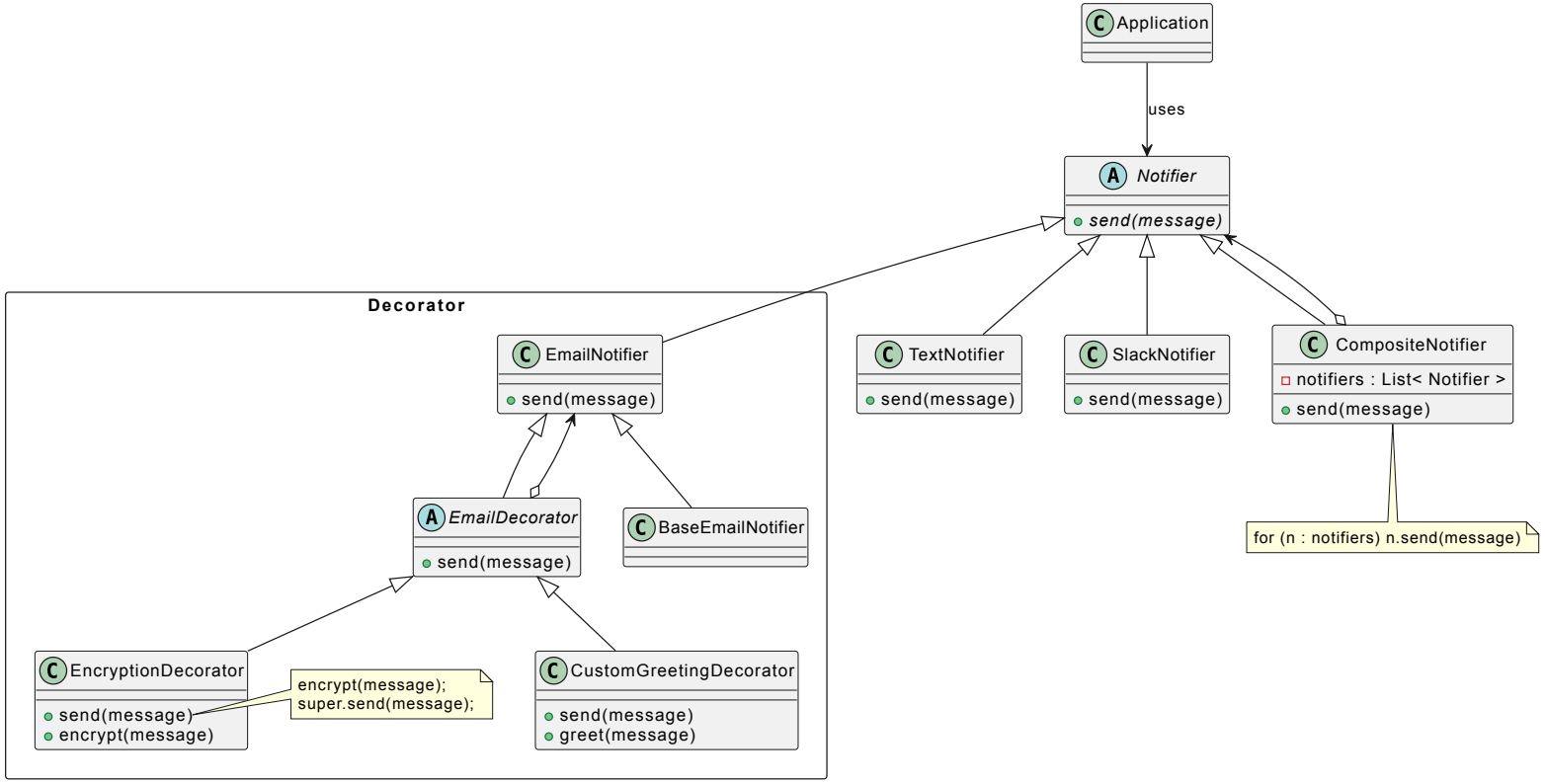


Power adapters for different countries

Image from "Dive into Design Patterns"

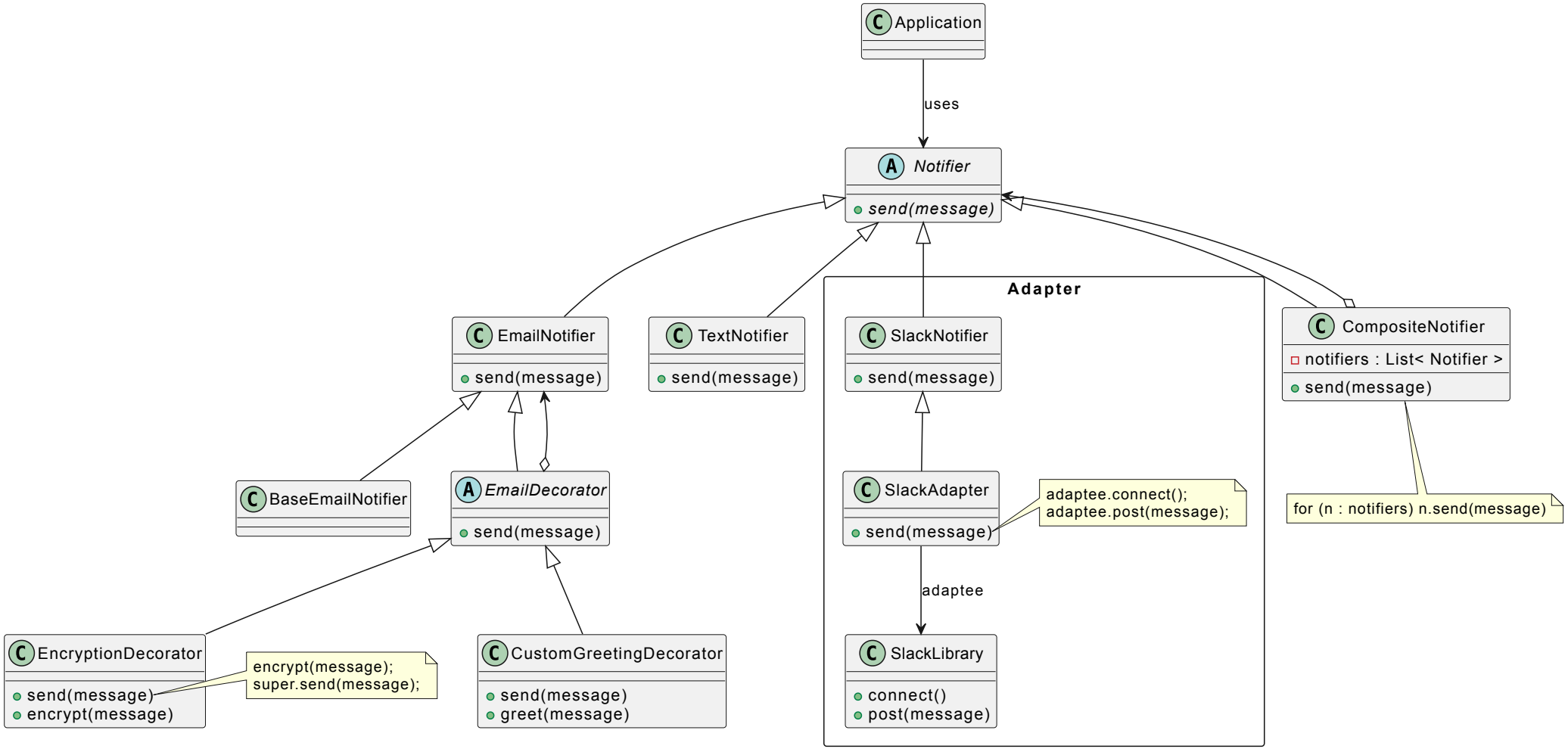
Design Challenge Revisited

- Application to send notifications over configurable channels



- We want to implement the `SlackNotifier` using a third-party library, whose interface does not fit our design

Design Challenge Improved



Implementation

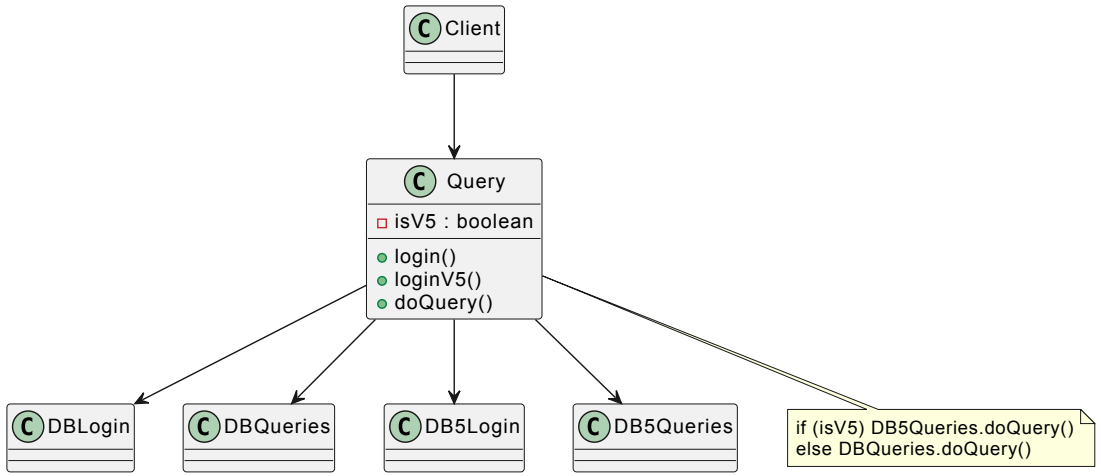
- Have at least 2 classes with incompatible interfaces
  - A service/library that cannot be changed
  - A client to use the service
- Declare a client interface `Target`
- Create the adapter class, implement the client interface
- Have the adapter reference the service or library
- Forward from adapter methods to service methods

```
// client uses a target service/library
public class Character {
  private int strength;

  private CharacterStore s;

  public void save() {
    s.save(this);
  }

  /* ... */
}

// target
public abstract class CharacterStore {
  public abstract void save(Character c);
}

// incompatible library
public class JSONStore {
  public void persist(JSON data) { /*...*/ }
}

// adapter
public class JSONCharacterStore extends CharacterStore {
  private JSONStore s;
  public void save(Character c) {
    s.persist(toJSON(c));
  }
  private JSON toJSON(Character c) {
    // convert
  }
}
```

Extract Adapter

Code Smell: A class adapts multiple versions of a library

Before: Conditional code



▶ After Refactoring

Discussion

Benefits

- Promotes the Single Responsibility Principle: separates the client interface from data conversion tasks
- Promotes the Open/Closed Principle: can introduce new types of adapters

Drawbacks

- Overall code complexity increases

Relations with Other Patterns

- **Adapter** vs **Decorator**
  - `Adapter` changes an interface
  - `Decorator` keeps an interface but adds behavior
- **Facade** often works with entire subsystem of objects, `Adapter` with 1 object