SE 350 - Object-Oriented Software Development

Software Design Patterns: Composite

Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Composite
- Understand the pattern

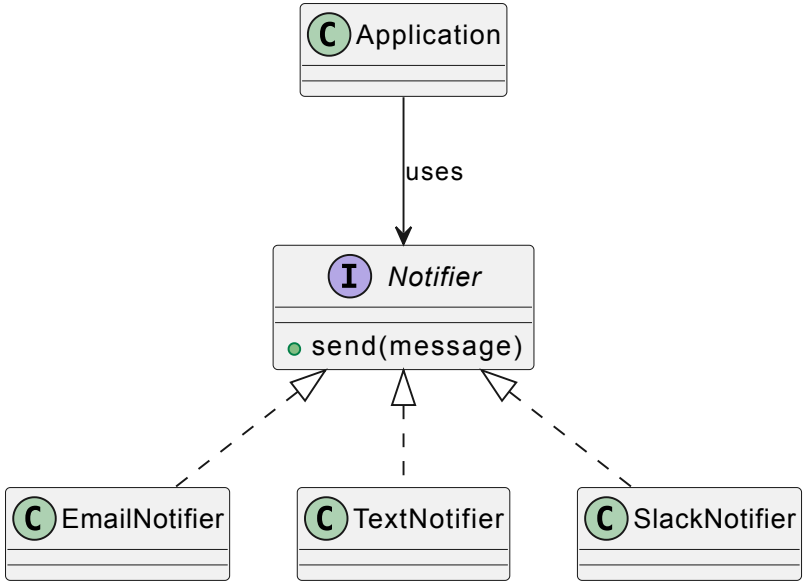
Design Challenge

- Requirement: application that can send notifications by email, text message, Slack

❓ Create an object-oriented design

Design Challenge

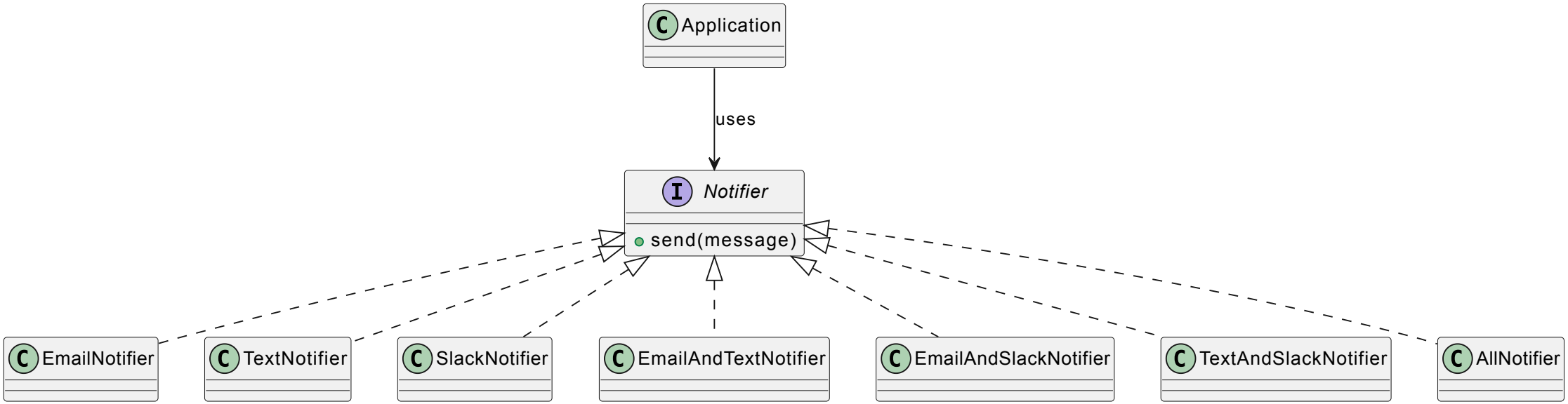
- Requirement: application that can send notifications by email, text message, Slack



- Requirement: users should be able to sign up for any combination of notifications

Design Challenge

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

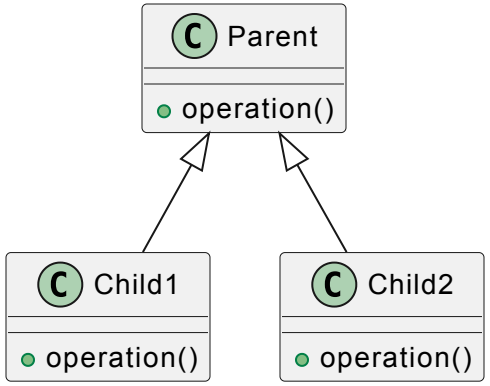


❓ What can be improved about this design?

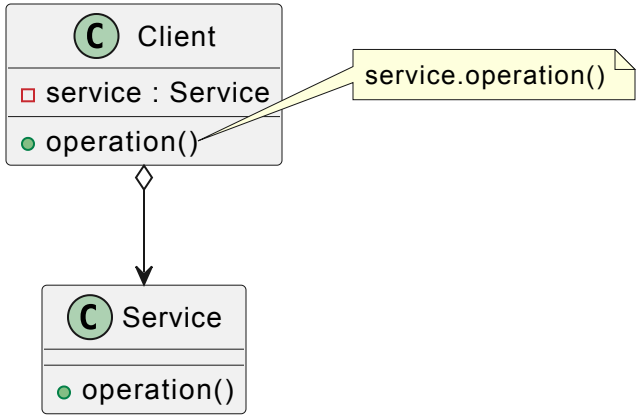
Design Challenge

- Inheritance often comes to mind first when extending behavior
- But inheritance has limitations
  - Inheritance is static
  - Subclasses have just one parent class
- Composition is a dynamic alternative to inheritance

Inheritance



Composition



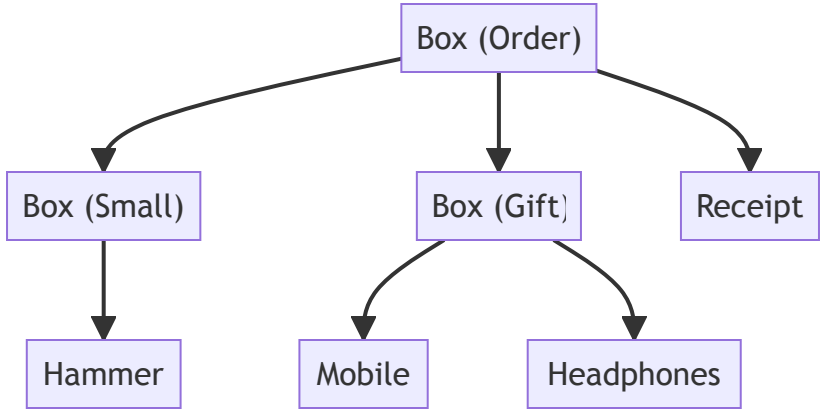
Introduction

Composite represents part-whole hierarchies in a tree structure

- Structural design pattern
- Lets clients treat individual objects and compositions uniformly

Problem illustration: want to pack products into boxes; a box can contain several products and smaller boxes

❓ How to represent a tree structure?



Overview

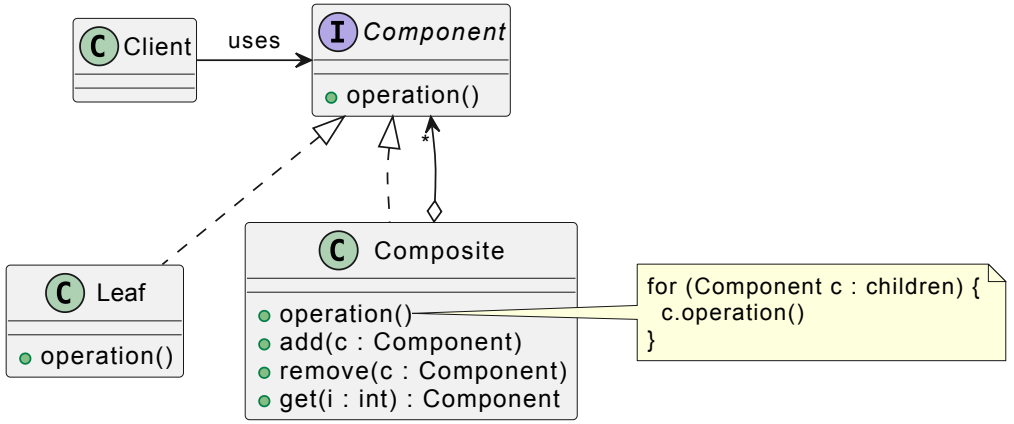
Problem

- Manipulate objects and compositions uniformly
- Hierarchically structure objects

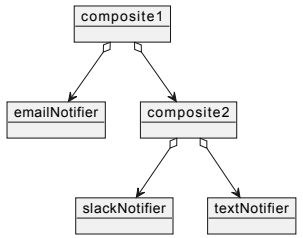
Intent

- Model objects in a tree structure
- Client does not distinguish between individual objects or composites
- Access objects and composites uniformly

Structure



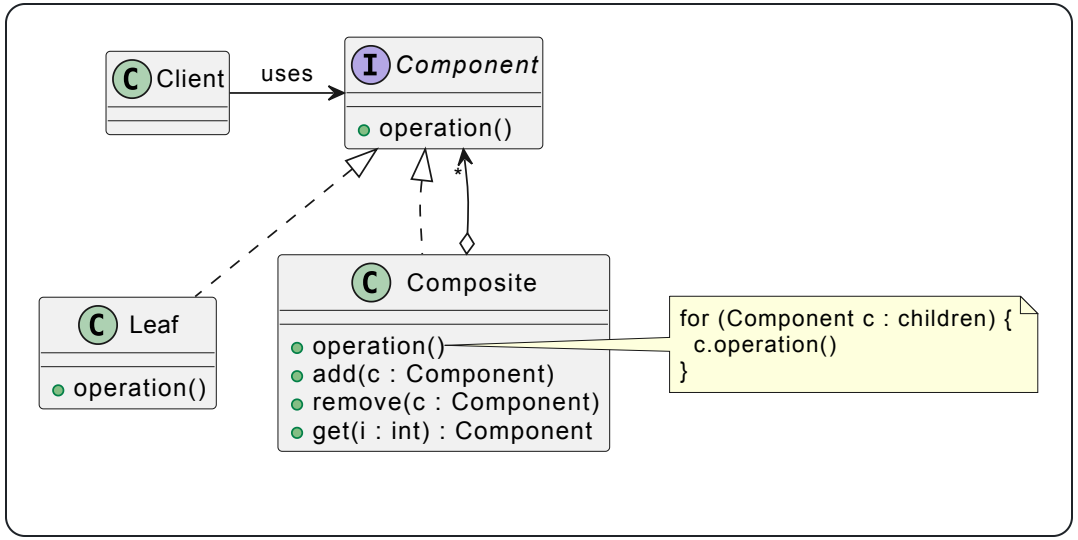
Example Object Diagram



Composite Participants

1. The `Component` interface describes operations that are common to both simple and complex elements of a tree

2. The `Leaf` is a basic element of a tree that does not have sub-elements; usually, leaf components do the bulk of the real work

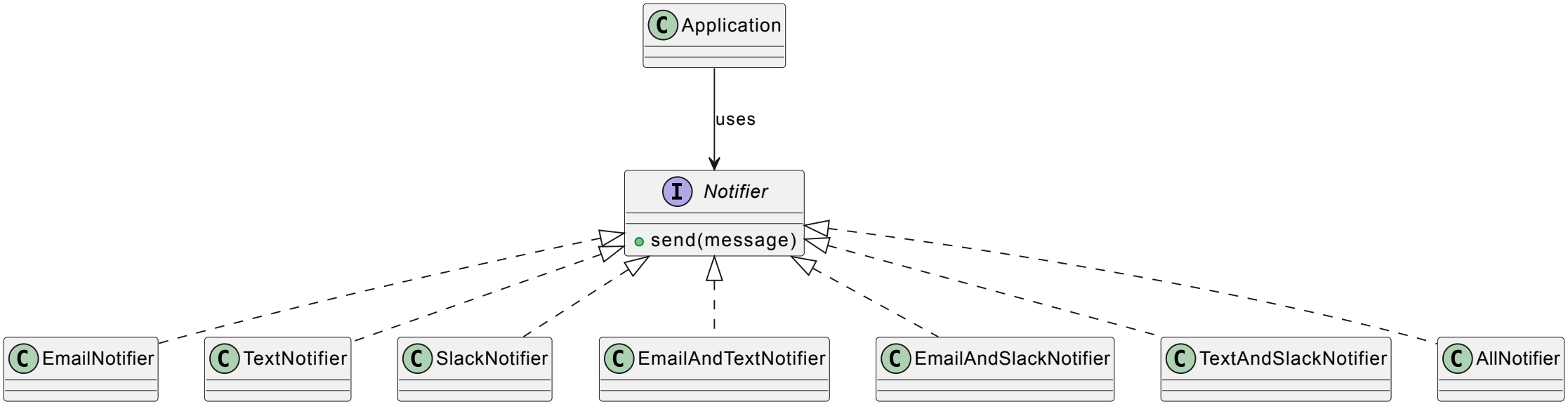


3. The `Composite` (a.k.a. `Container`) has sub-elements and delegates work

4. The `Client` works uniformly with all elements through the `Component` interface

Design Challenge Revisited

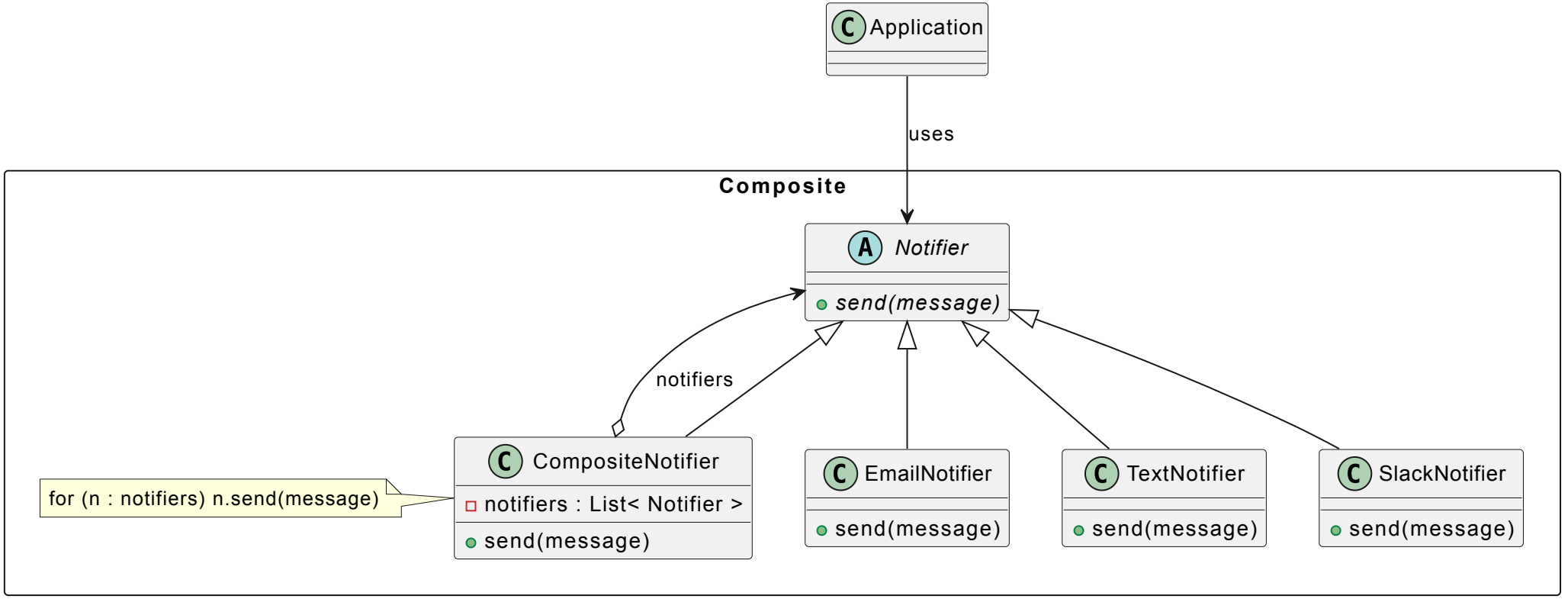
- Requirement: application that can send notifications by email, text message, Slack
- Requirement: users should be able to sign up for any combination of notifications



❓ What can be improved about this design?

Design Challenge Improved

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



Implementation

- Make sure that the relevant parts of your app can be represented as a tree structure
- Declare the `Component` interface
- Create `Leaf` classes for all simple elements; multiple different `Leaf` classes are possible
- Create a container class to represent complex elements; store children in a `List`, `Array`, or some other container
- Define methods for adding and removing children

```
// component
public interface Shape {
  int getX();
  int getY();
  void paint(java.awt.Graphics graphics);
}

// leafs
public class Dot implements Shape { /*...*/ }
public class Circle implements Shape { /*...*/ }

// composite
public class Figure implements Shape {
  private List<Shape> children = new ArrayList<>();
  public Figure(Shape... components) {
    children.addAll(components);
  }
  public void add(Shape c) {
    children.add(c);
  }
  public void remove(Shape c) {
    children.remove(c);
  }
  public void paint(java.awt.Graphics graphics) {
    for (Shape c : children) c.paint(graphics);
  }
}
```

Composite Variations

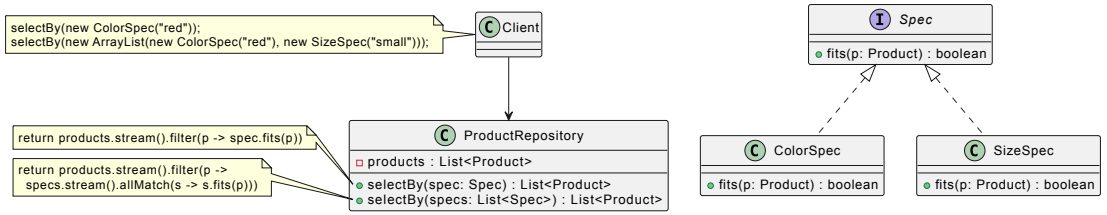
- **Explicit parent references** in children: can help traversal and management of the tree structure
- **Sharing components**: reduce memory usage, but difficult to combine with parent references
- **Maximize the `Component` interface**: since `Component` defines the common of `Leaf` and `Composite`, define as many operations as possible (violates Interface Segregation Principle)
- **Where to declare `add` and `remove`**: in `Component` violates Liskov Substitution Principle (`Leaf` implements empty); when in `Composite` loose uniform handling of all components
- **Child ordering**

Replace One/Many Distinction with Composite

Code Smell: A class processes single and multiple objects using separate pieces of code

Before: Client needs to distinguish multiple methods

▶ After Refactoring

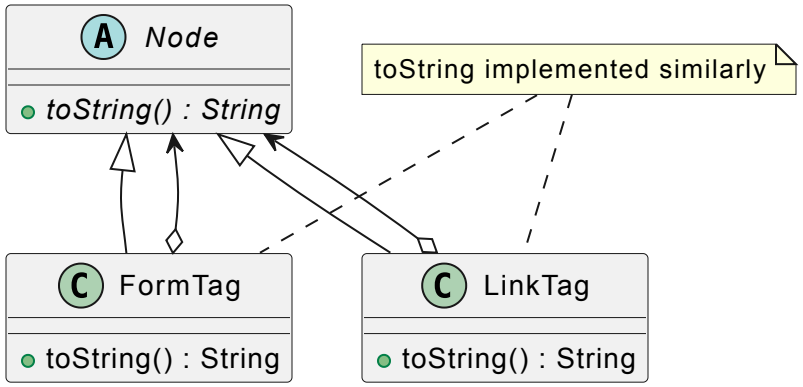


- What if `List<Spec>` should be interpreted as disjunction instead?

Extract Composite

Code Smell: Subclasses in a hierarchy implement the same composite

Before: Duplicate composite code



▶ After Refactoring

Discussion

Benefits

- Clients work with tree structures more conveniently
- Promotes the Open/Closed Principle: can introduce new `Leaf` and `Composite` classes without breaking existing code

Drawbacks

- Can be difficult to provide a common `Component` interface when concrete classes differ too much
- May violate Interface Segregation Principle

Relations with Other Patterns

- Can use **Builder** to create complex trees
- Can use **Iterator** to traverse trees
- Can use **Visitor** to execute an operation over entire tree
- Can use **Chain of Responsibility** to pass a request from leaf up through all parents to root