Design Patterns

Ahmed Mamdouh Mohamed

March 2022

References

- “Dive into design patterns” – Alexander Shvets.
  https://refactoring.guru/
- “Head First Design Patterns”
- https://app.pluralsight.com/library/courses/csharp-solid-principles

What is a design pattern?

- Typical solutions to commonly occurring problems in software design
- Pre-made blue prints that you can customize to solve a recurring design problem in your code
- Not a specific piece of code, but a general concept for solving a particular problem.
- Follow the pattern details and implement a solution that suits the realities of your own program.

What does the pattern consist of ?

- Intent of the pattern briefly describes both the problem and the solution.
- Motivation further explains the problem and the solution the pattern makes possible.
- Structure of classes shows each part of the pattern and how they are related.
- Code example in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

Who invented Patterns?

When a solution gets repeated over and over in various projects, someone eventually puts a name to it and describes the solution in detail.

It was discovered, not invented

Why Should I Learn Patterns?

- Teaches you how to solve all sorts of problems using principles of object-oriented design.

- Define a common language that you and your teammates can use to communicate more efficiently

Classification of patterns

- Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code.

- Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

- Behavioral patterns take care of effective communication and the assignment of responsibilities between objects.

Features of Good Design

- Code reuse.

- Extensibility

  Change is the only constant thing in a programmer's life

Design Principles

- **Encapsulate What Varies**

  Identify the aspects of your application that vary and separate them from what stays the same.

- **Program to an Interface, not an Implementation**

  Depend on abstractions, not on concrete classes.

- **Favor Composition Over Inheritance**

Encapsulate What Varies

- Encapsulation on a method level

```
1   method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4       total += item.price * item.quantity
5
6     if (order.country == "US")
7       total += total * 0.07 // US sales tax
8     else if (order.country == "EU"):
9       total += total * 0.20 // European VAT
10
11    return total
```

tax calculation code is mixed with the rest of the method's code

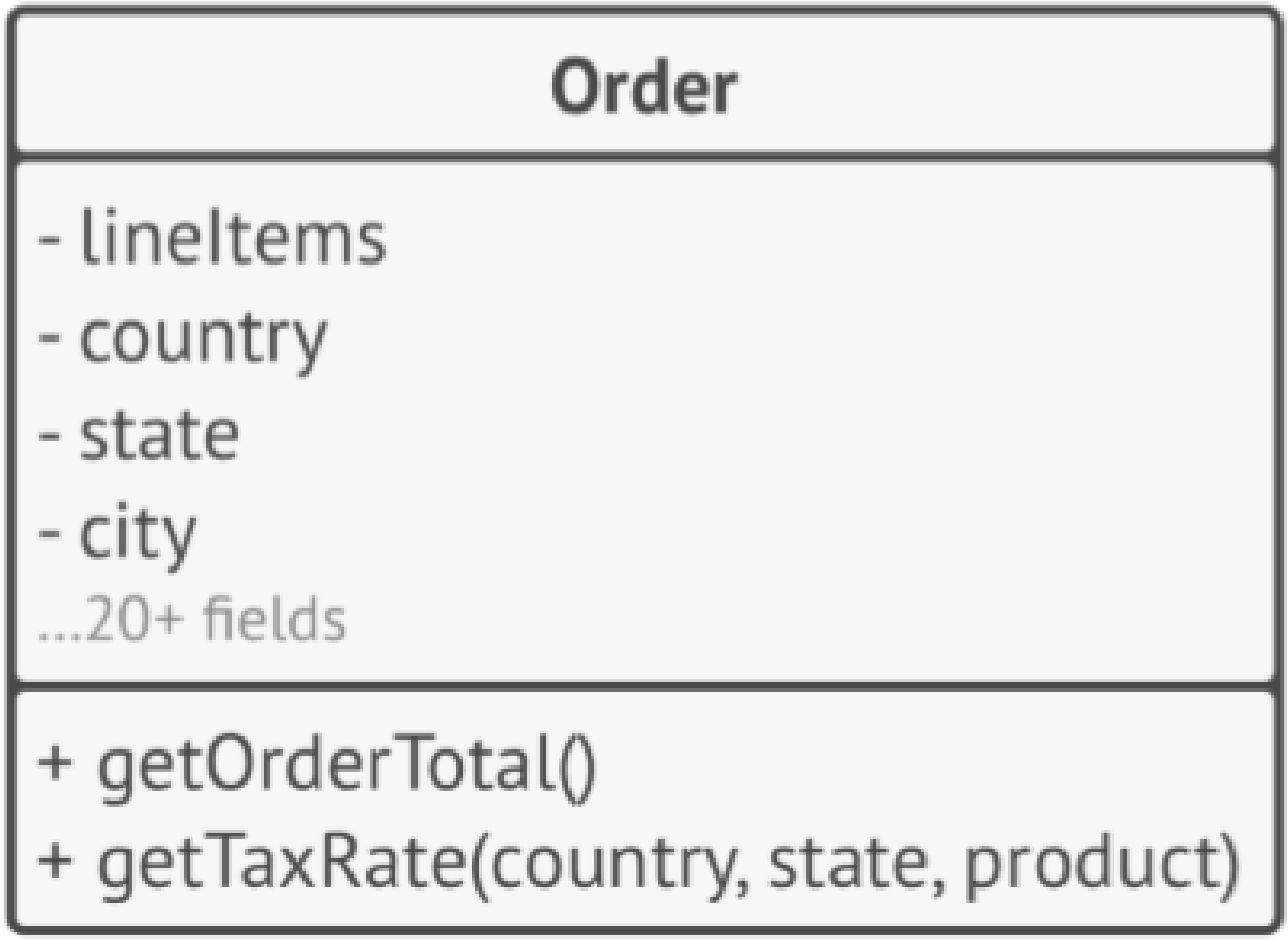
Encapsulate What Varies (cont.)

- Encapsulation on a **method** level

```
1   method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4       total += item.price * item.quantity
5
6     total += total * getTaxRate(order.country)
7
8     return total
9
10  method getTaxRate(country) is
11    if (country == "US")
12      return 0.07 // US sales tax
13    else if (country == "EU")
14      return 0.20 // European VAT
15    else
16      return 0
```

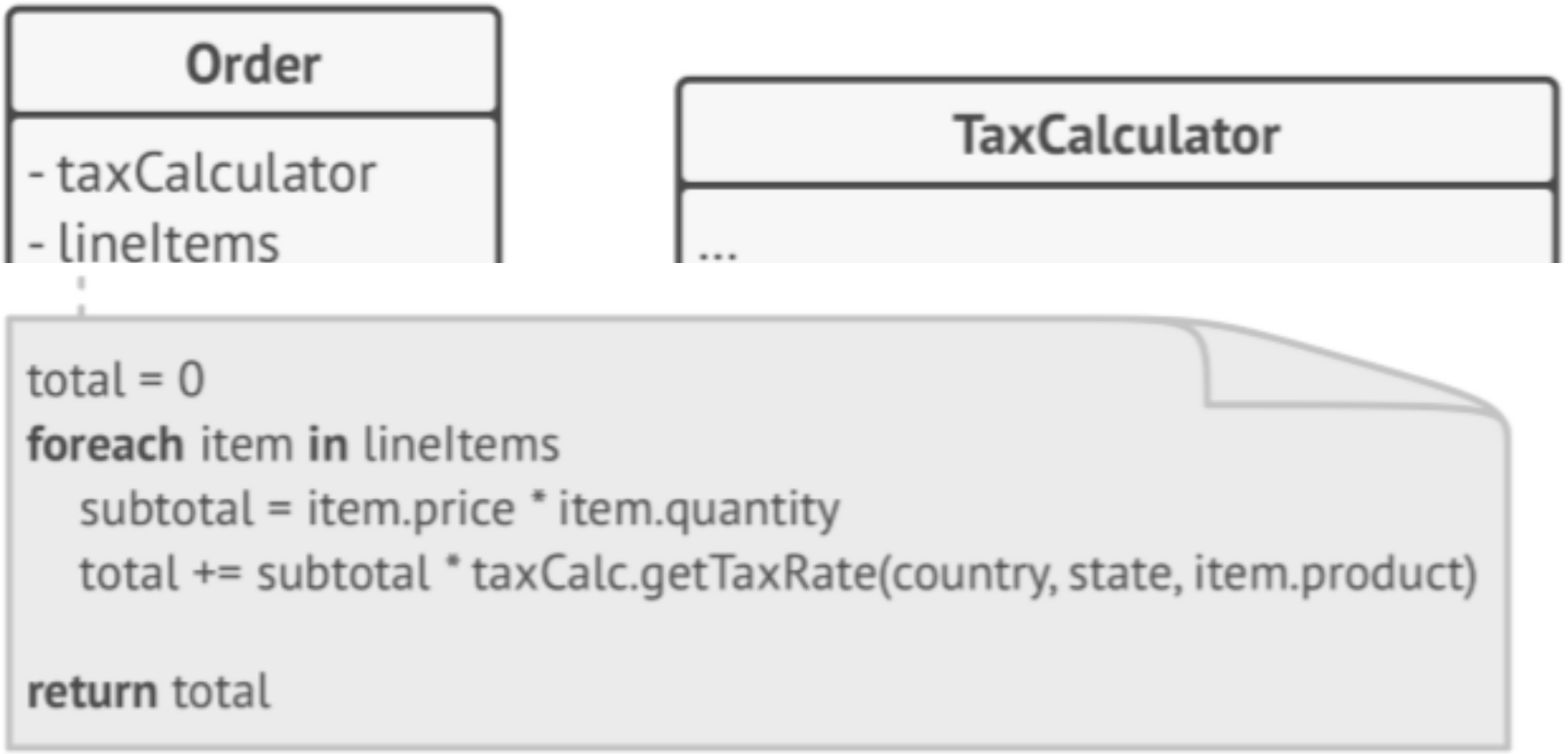
Encapsulate What Varies (cont.)

- **Encapsulation on a class level**



Encapsulate What Varies (cont.)

- Encapsulation on a **class** level

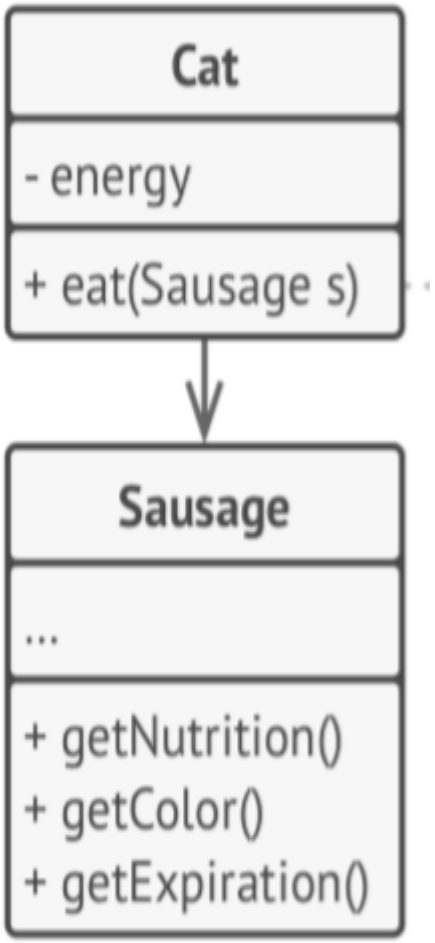


tax calculation is hidden from the order class

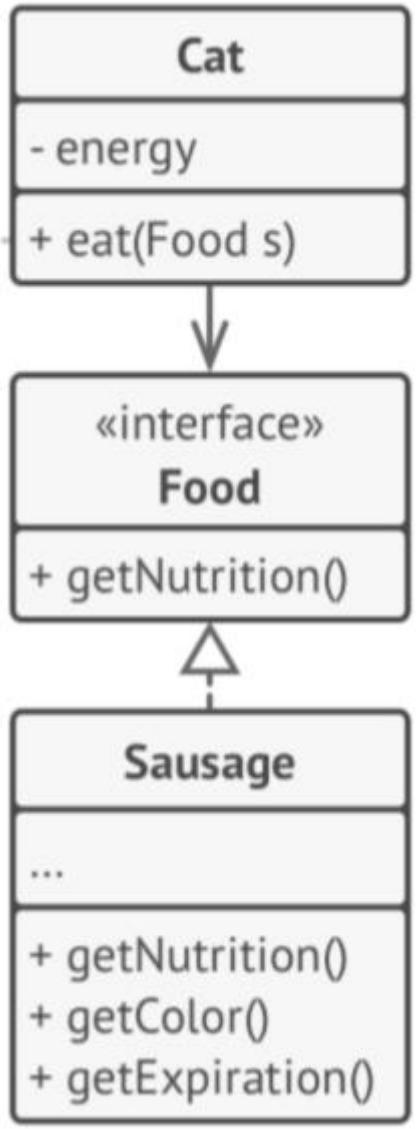
Design Principles

- Encapsulate What Varies
- Program to an Interface, not an Implementation
- Favor Composition Over Inheritance

Program to an Interface, not an Implementation



Program to an Interface, not an Implementation



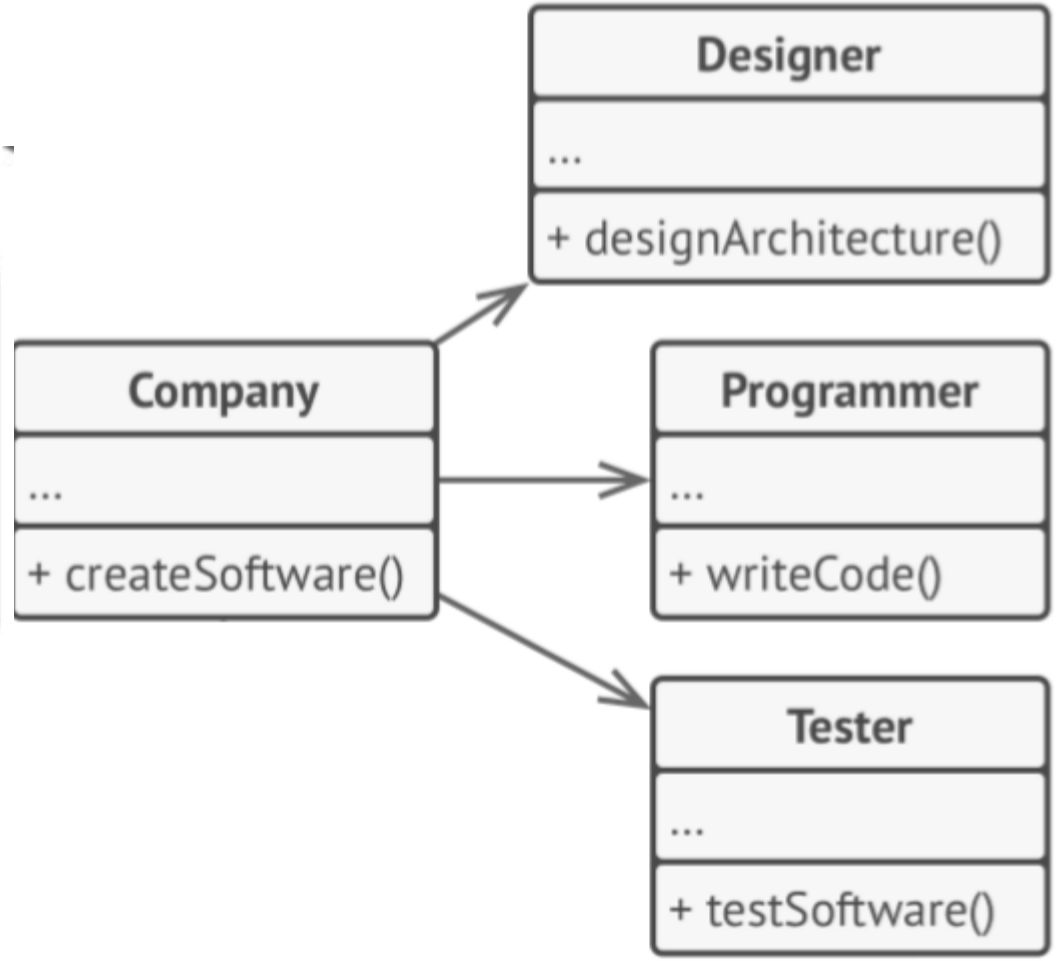
More flexible,
yet more complicated

Program to an Interface, not an Implementation

- Determine what exactly one object needs from the other: which methods does it execute?
- Describe these methods in a new interface or abstract class.
- Make the class that is a dependency implement this interface.
- Now make the second class dependent on this interface rather than on the concrete class

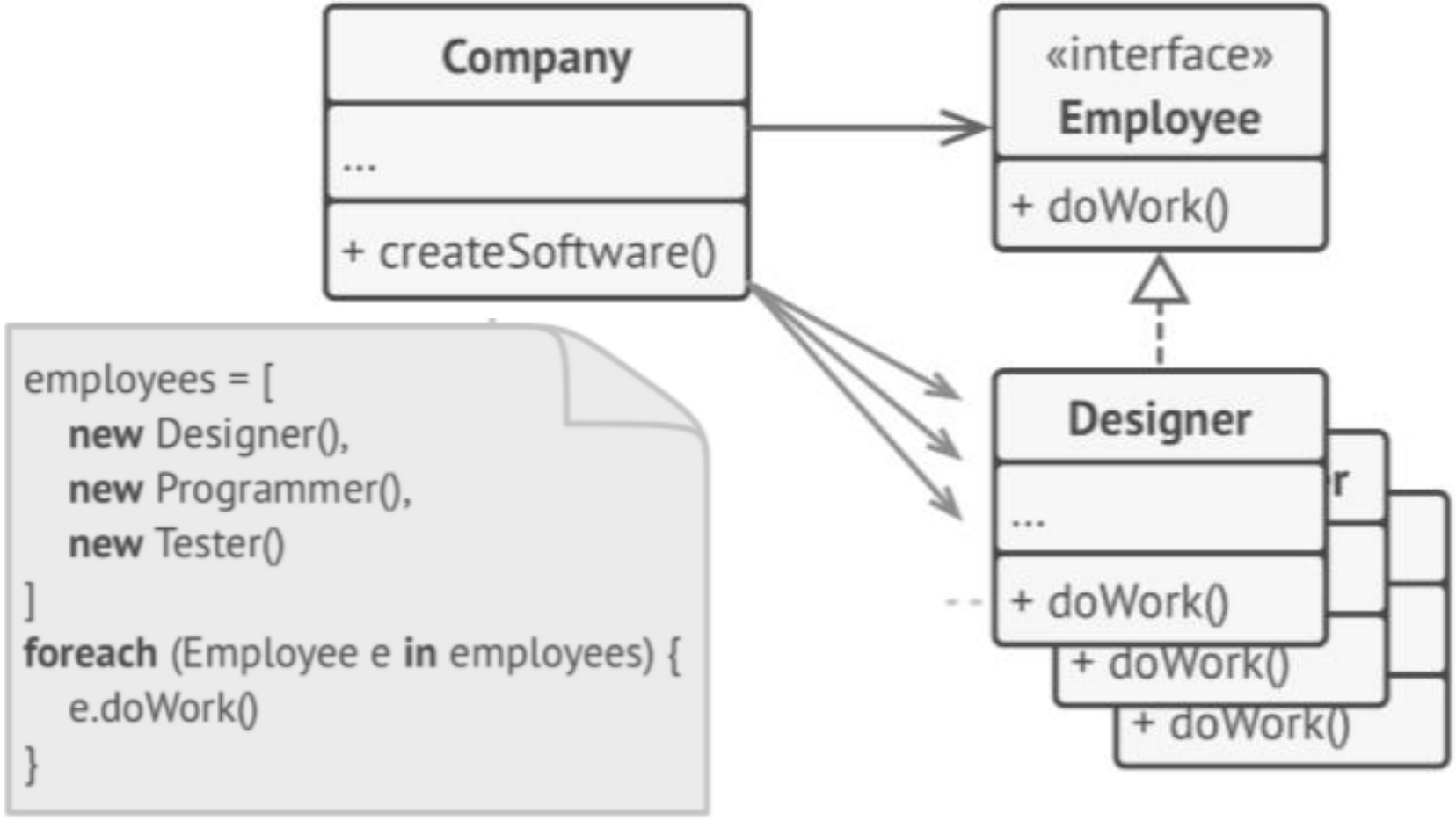
Program to an Interface, not an Implementation

```
Designer d = new Designer()
d.designArchitecture()
Programmer p = new Programmer()
p.writeCode()
Tester t = new Tester()
t.testSoftware()
```



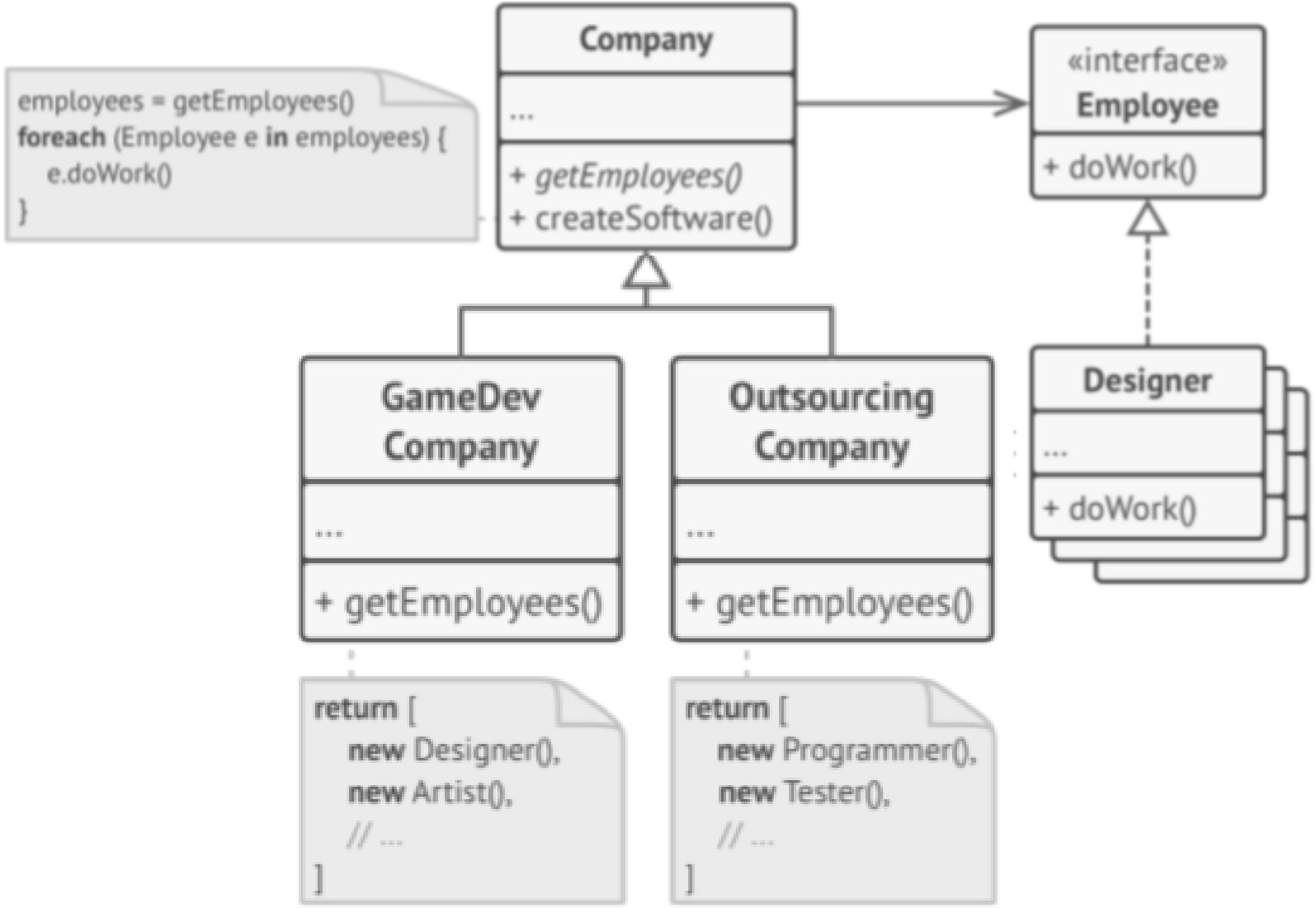
All classes are tightly coupled

Program to an Interface, not an Implementation



Company class still depends on the concrete employee classes

Program to an Interface, not an Implementation



Design Principles

- Encapsulate What Varies
- Program to an Interface, not an Implementation
- Favor Composition Over Inheritance

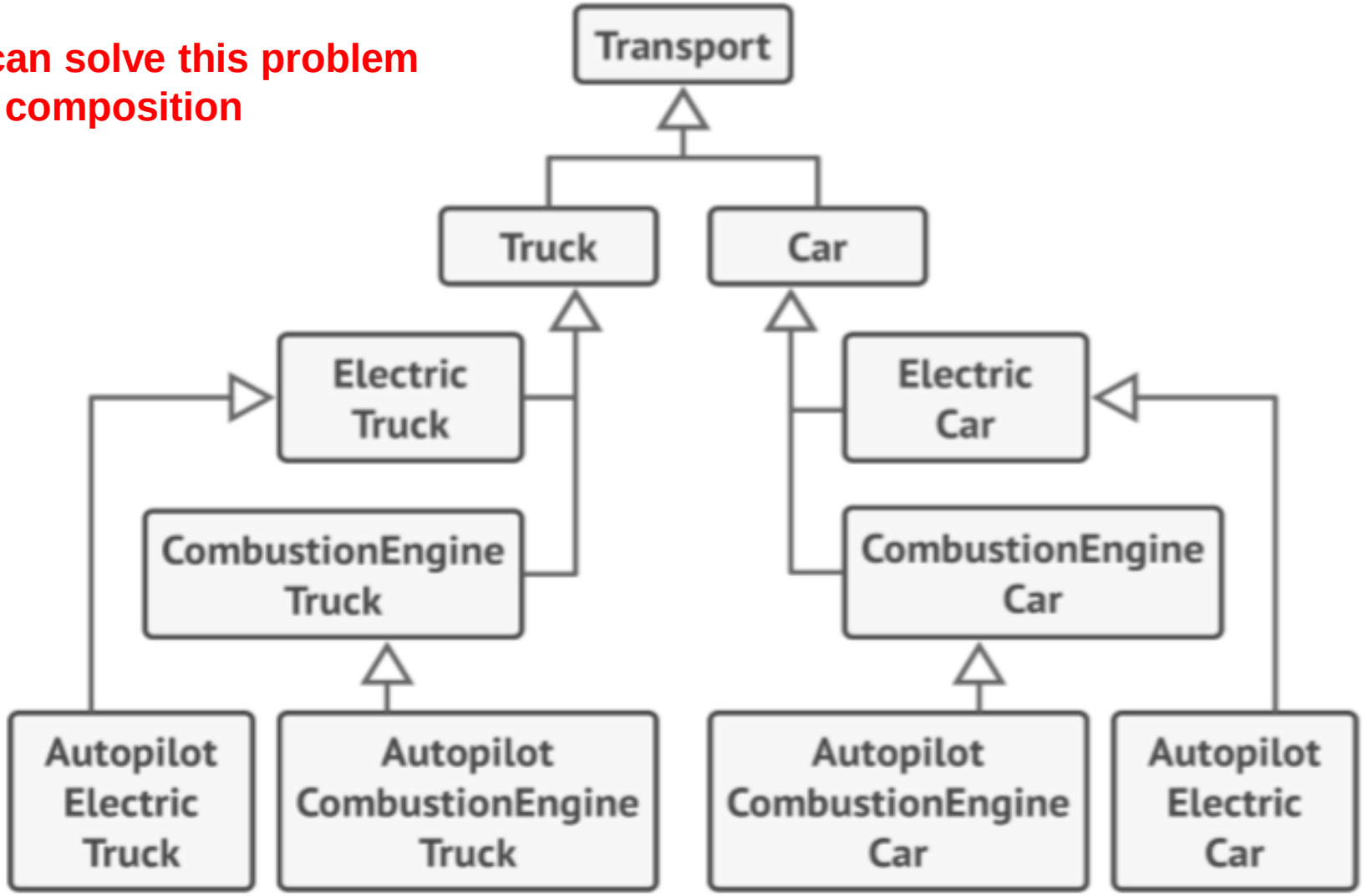
Favor Composition Over Inheritance

Problems with inheritance:

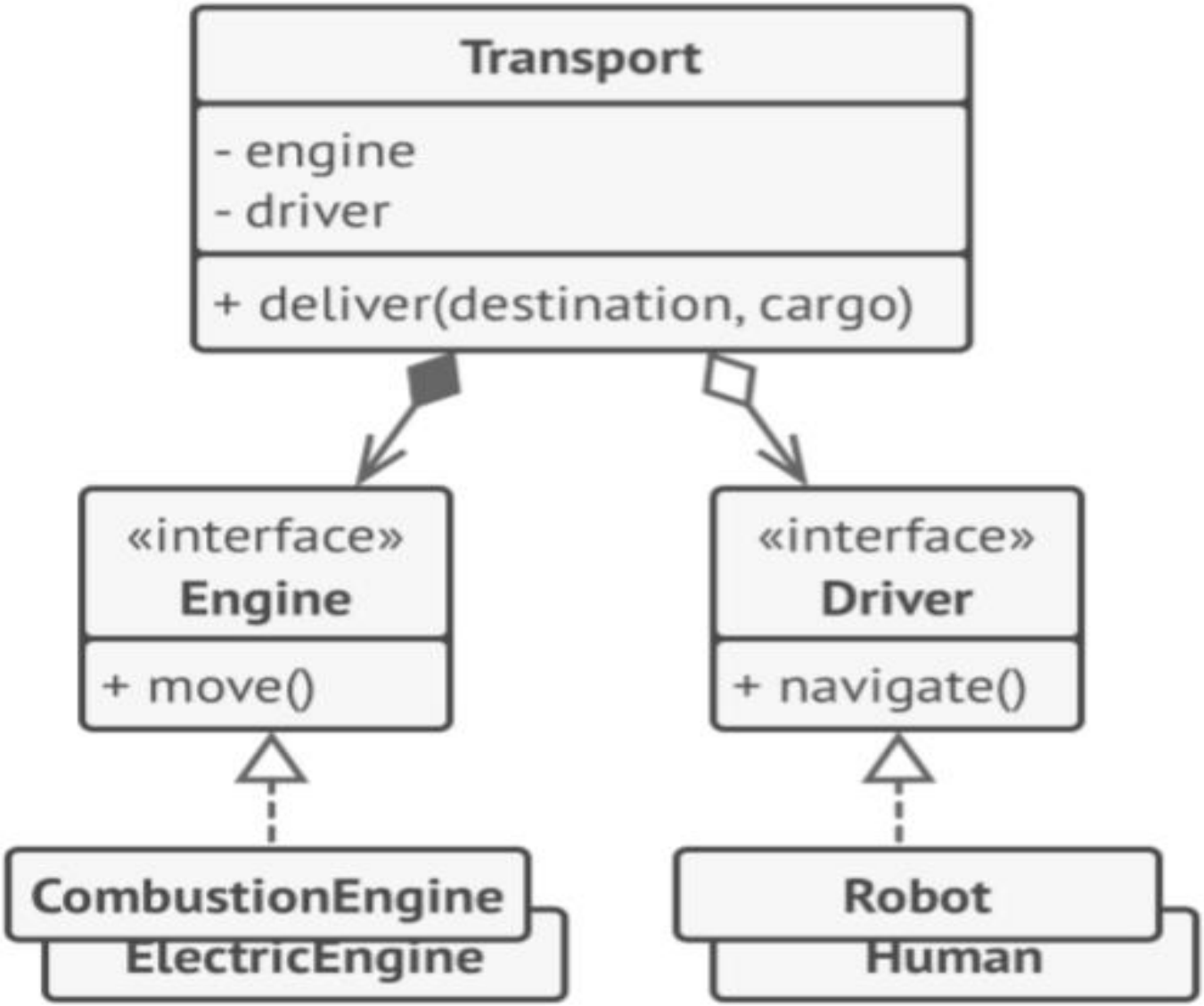
- A subclass can't reduce the interface of the super class.
- When overriding methods you need to make sure that the new behavior is compatible with the base one.
- Inheritance breaks encapsulation of the super class
- Subclasses are tightly coupled to super classes
- Trying to reuse code through inheritance can lead to creating parallel inheritance hierarchies

Favor Composition Over Inheritance

We can solve this problem with composition



Favor Composition Over Inheritance



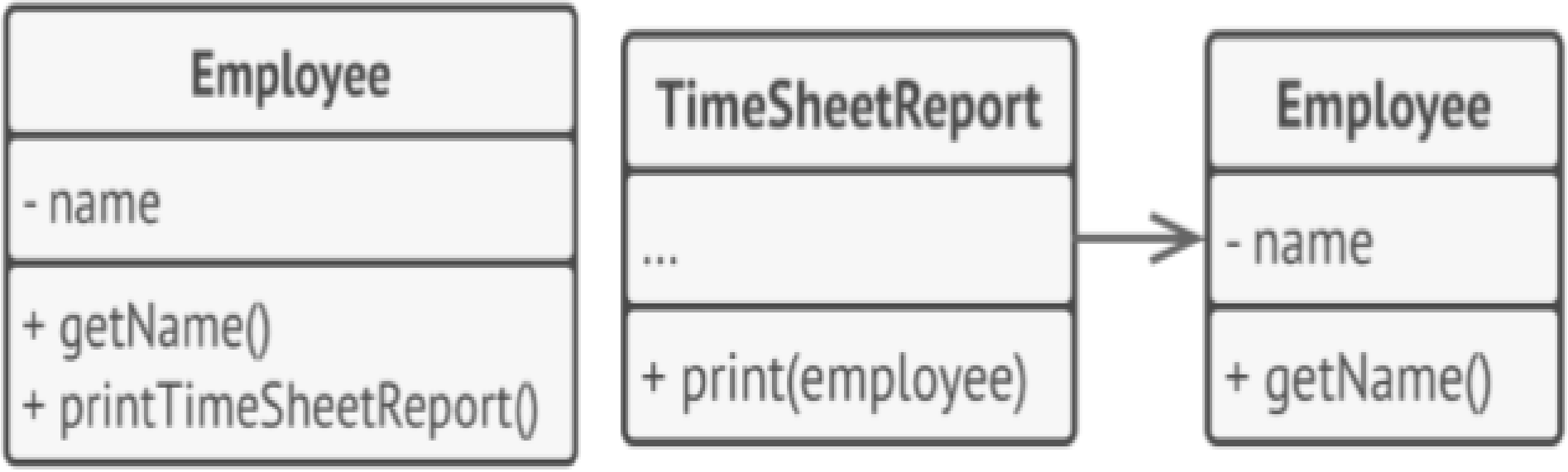
SOLID Principles

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

"A class should have just one reason to change"

The main goal of this principle is reducing complexity



Single Responsibility Principle

- Tight coupling vs Loose coupling
- Separation of concerns
- Cohesion

Logger Example

SOLID Principles

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Open/Closed Principle

Classes should be open for extension but closed for modification

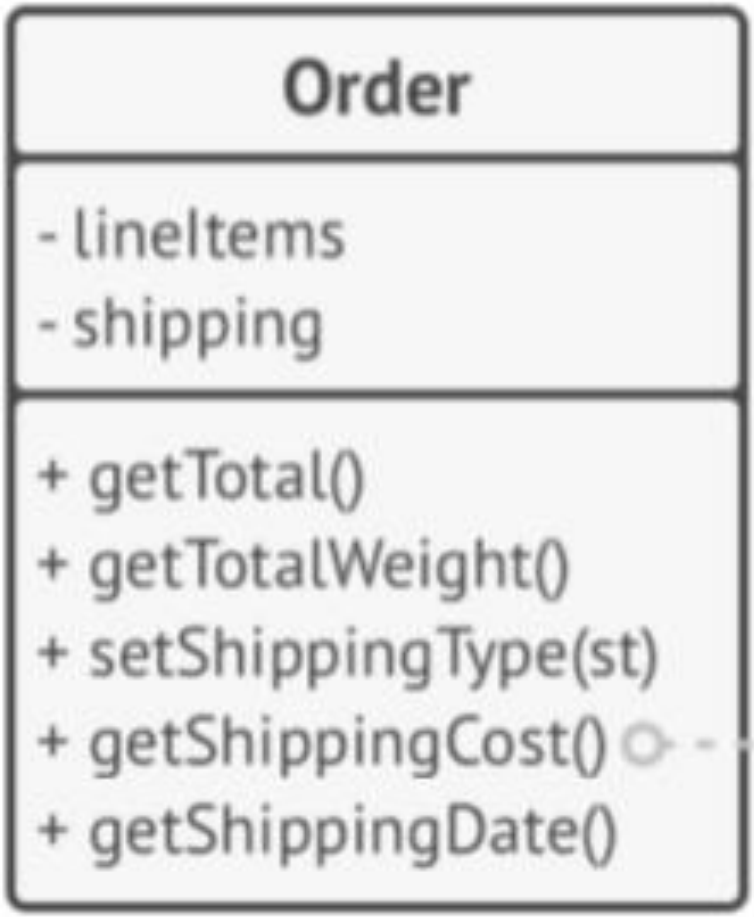
Why code should be closed to modification?

Less likely to introduce new bugs in code we don't touch

Less likely to break dependent code when we don't have to deploy updates

What if we have bugs in code?

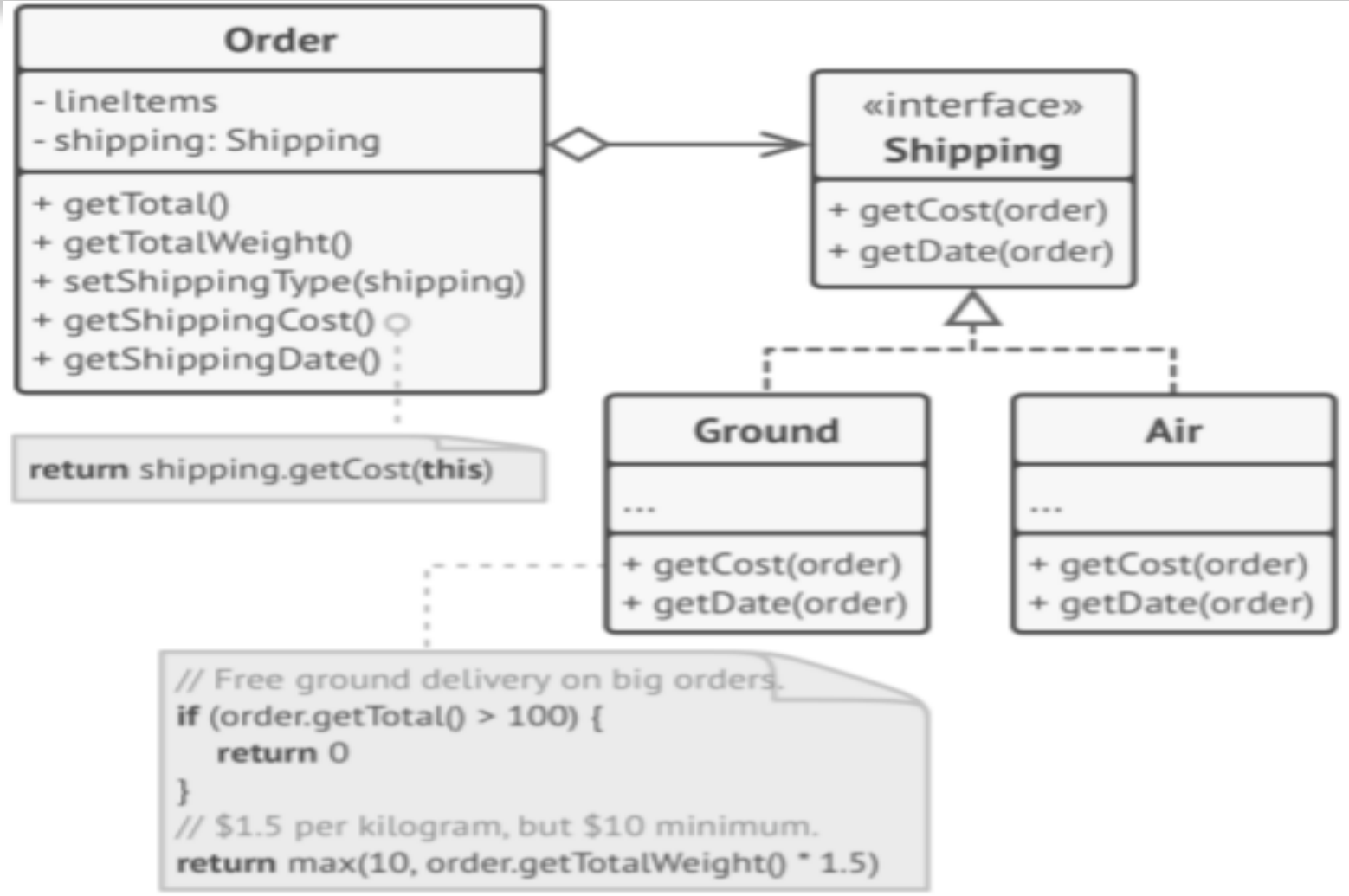
Open/Closed Principle




```
if (shipping == "ground") {
    // Free ground delivery on big orders.
    if (getTotal() > 100) {
        return 0
    }
    // $1.5 per kilogram, but $10 minimum.
    return max(10, getTotalWeight() * 1.5)
}

if (shipping == "air") {
    // $3 per kilogram, but $20 minimum.
    return max(20, getTotalWeight() * 3)
}
```

you have to change the Order class whenever you add a new shipping method to the app

Open/Closed Principle



Adding a new shipping method doesn't require changing existing classes

Open/Closed Principle

- Extensible code, is Abstract
- Abstraction adds complexity
- Balance Abstraction and concreteness (What & How)
- Predict where variation is needed and apply abstraction as needed

Open/Closed Principle

How can we predict future changes

Start concrete

Modify the code the first time or two

By the third modification, consider making the code open to extension for the axis of change

Typical approaches to OCP

- **Parameters**

Extremely Concrete

```
public class DoOneThing
{
    public void Execute()
    {
        Console.WriteLine("Hello world.");
    }
}
```

Parameter-Based Extension

```
public class DoOneThing
{
    public void Execute(string message)
    {
        Console.WriteLine(message);
    }
}
```

Typical approaches to OCP

- **Inheritance**

Inheritance-based Extension

```
public class DoOneThing
{
    public virtual void Execute()
    {
        Console.WriteLine("Hello world.");
    }
}
public class DoAnotherThing
{
    public override void Execute()
    {
        Console.WriteLine("Goodbye world!");
    }
}
```

Typical approaches to OCP

- **Composition / Injection**

Composition/Injection Extension

```
public class DoOneThing
{
    private readonly MessageService _messageService;
    public DoOneThing(MessageService messageService)
        => _messageService = messageService;

    public void Execute()
    {
        Console.WriteLine(_messageService.GetMessage());
    }
}
```

SOLID Principles

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Liskov Substitution Principle

When extending a class, remember that you should be able to pass objects of the subclass in place of objects of the parent class without breaking the client code.

- The subclass should remain compatible with the behavior of the superclass.

- When overriding a method, extend the base behavior rather than replacing it with something else entirely

Detecting LSP Violations

- Type checking with is / as in polymorphic code

Type Checking (Corrected)

```
foreach(var employee in employees)
{
    employee.Print();
}

// OR

foreach(var employee in employees)
{
    Helpers.PrintEmployee(employee);
}
```

Detecting LSP Violations

- Not Implemented Exceptions

Not Implemented Exceptions

```
public class SmtpNotificationService : INotificationService
{
    public void SendEmail(string to, string from,
                          string subject, string body)
    {
        // actually send email here
    }

    public void SendText(string SmsNumber, string message)
    {
        throw new NotImplementedException();
    }
}
```

Liskov Substitution Principle

- ✓ A subclass shouldn’t strengthen pre-conditions.
- ✓ A subclass shouldn’t weaken post-conditions
- ✓ A subclass shouldn’t change values of private fields of the superclass

SOLID Principles

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

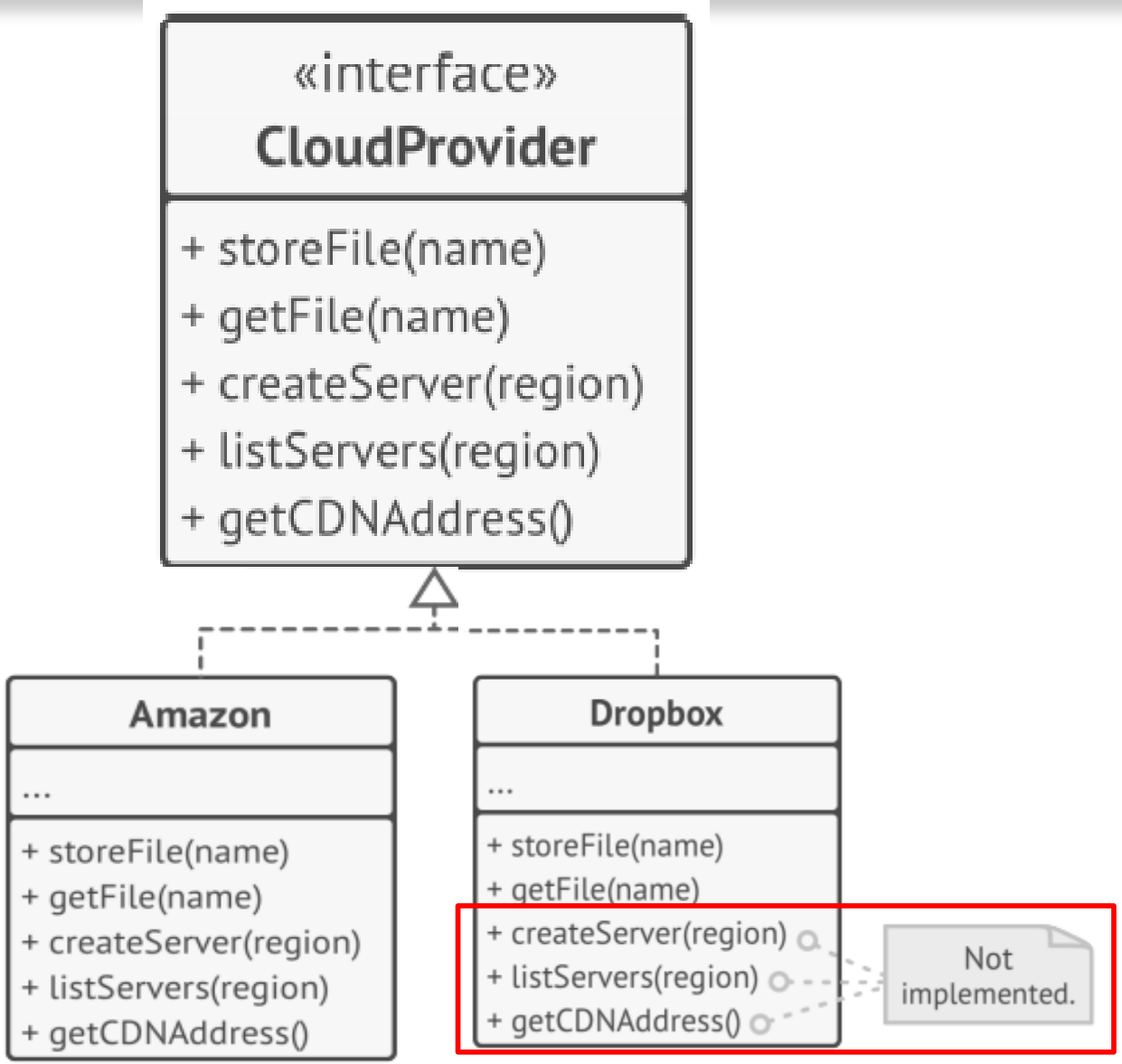
Interface Segregation Principle

Clients shouldn't be forced to depend on methods they do not use

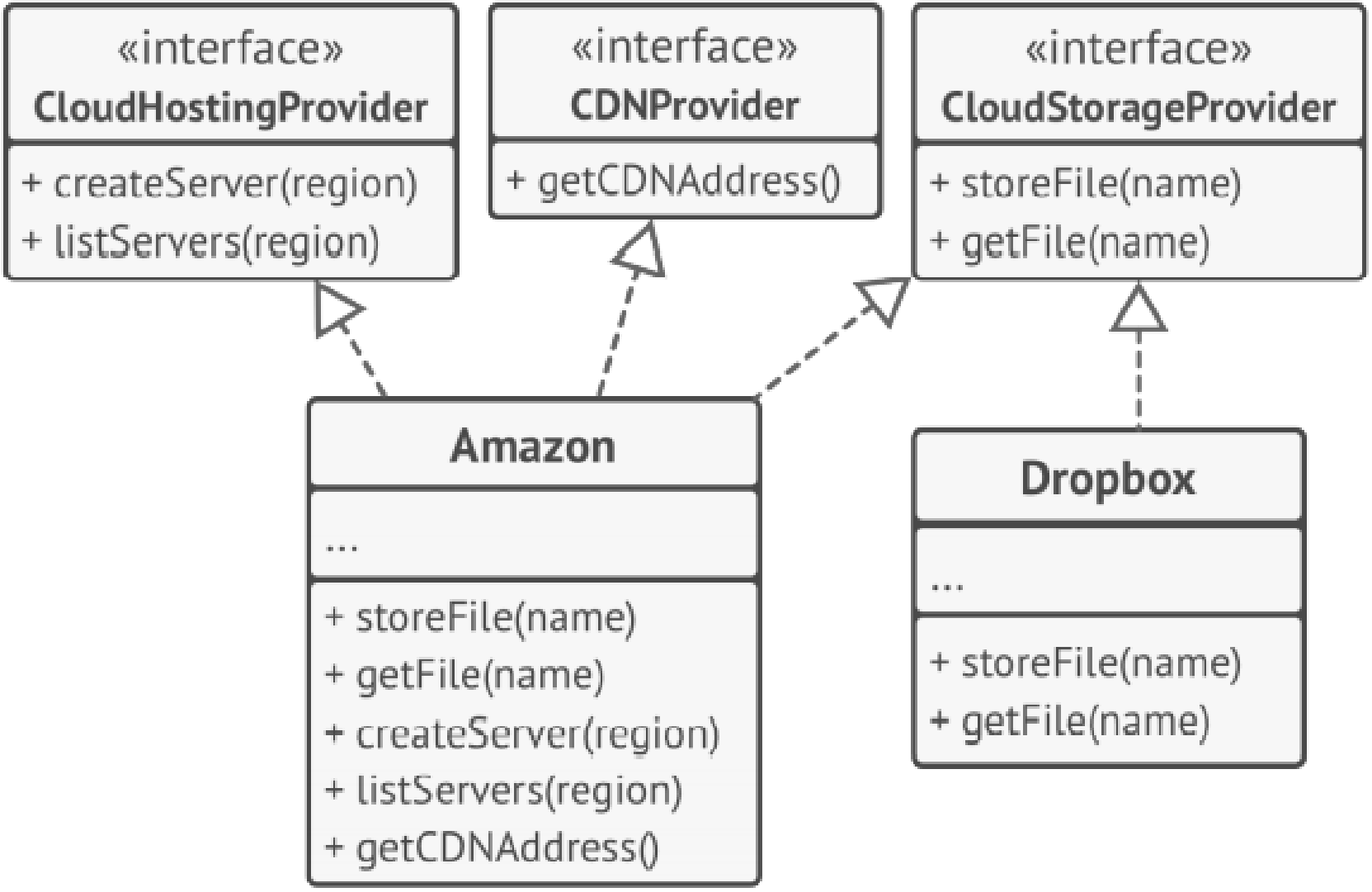
- You should break down "fat" interfaces into more granular and specific ones

- Clients should implement only those methods that they really need.

Interface Segregation Principle



Interface Segregation Principle



Detecting ISP Violations

- Large interfaces
- Not implemented exceptions
- Code uses a small subset of a larger interface

Detecting ISP Violations

Split It Up

```
public interface IEmailNotificationService
{
    void SendEmail(string to, string from,
                   string subject, string body);
}

public interface ITextNotificationService
{
    void SendText(string SmsNumber, string message);
}
```

What about legacy code that's coupled to the original interface?!

```
public interface INotificationService :
                                IEmailNotificationService,
                                ITextNotificationService
{
}
```

SOLID Principles

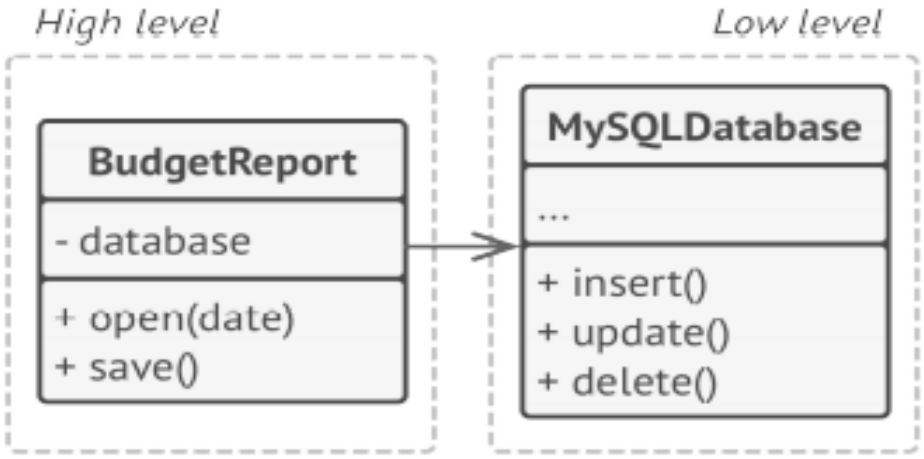
- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Dependency Inversion Principle

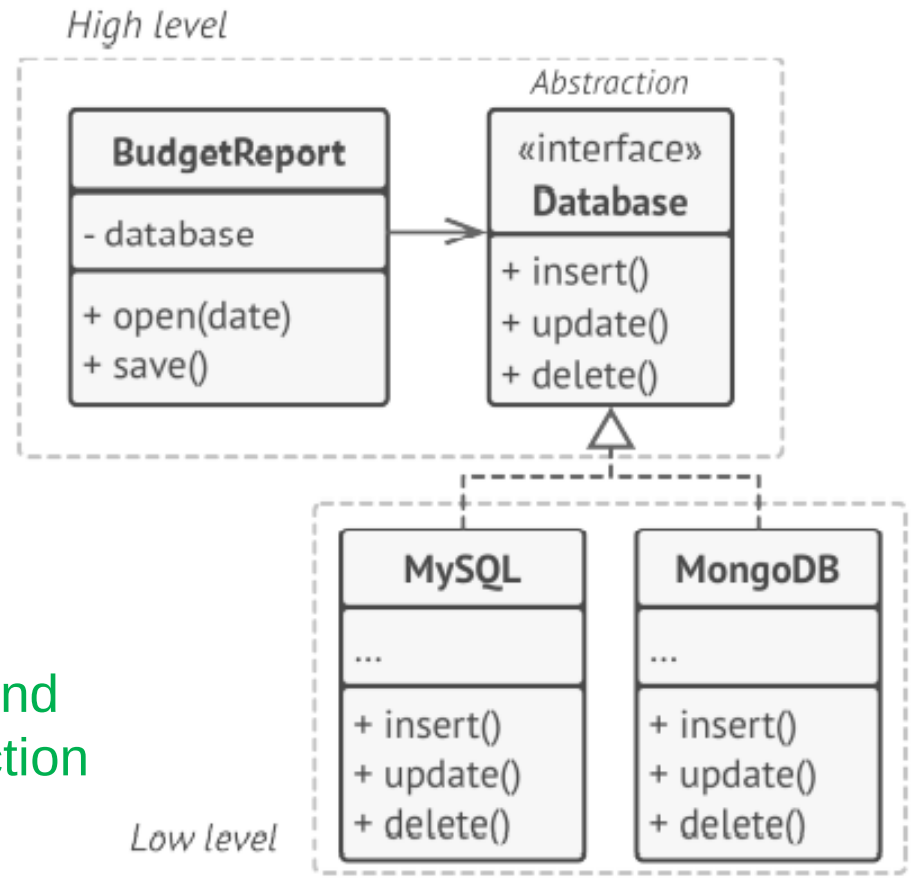
High-level classes shouldn’t depend on low-level classes. Both should depend on abstractions

Abstractions shouldn’t depend on details. Details should depend on abstractions

Dependency Inversion Principle



high-level class depends
on a low-level class



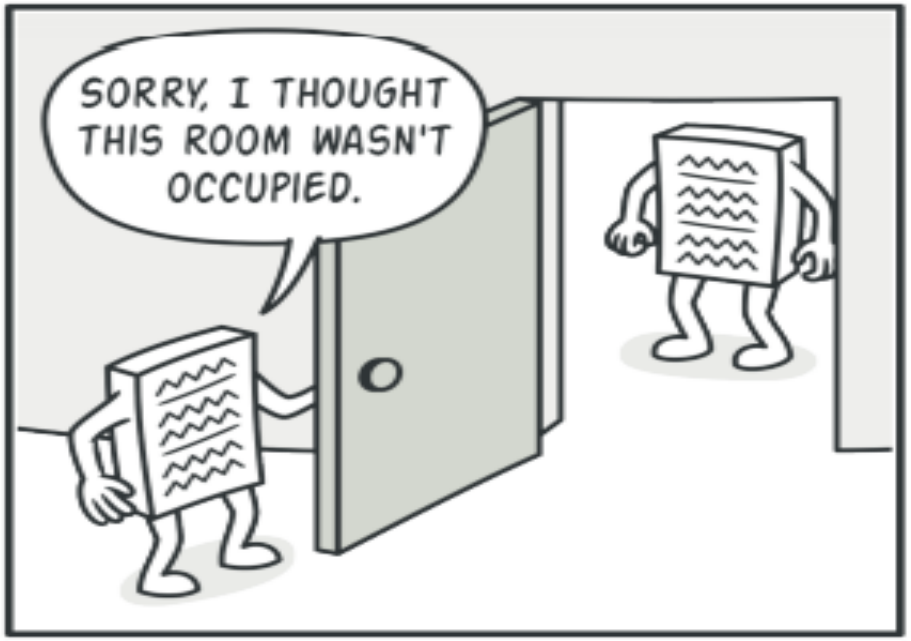
low-level classes depend
on a high-level abstraction

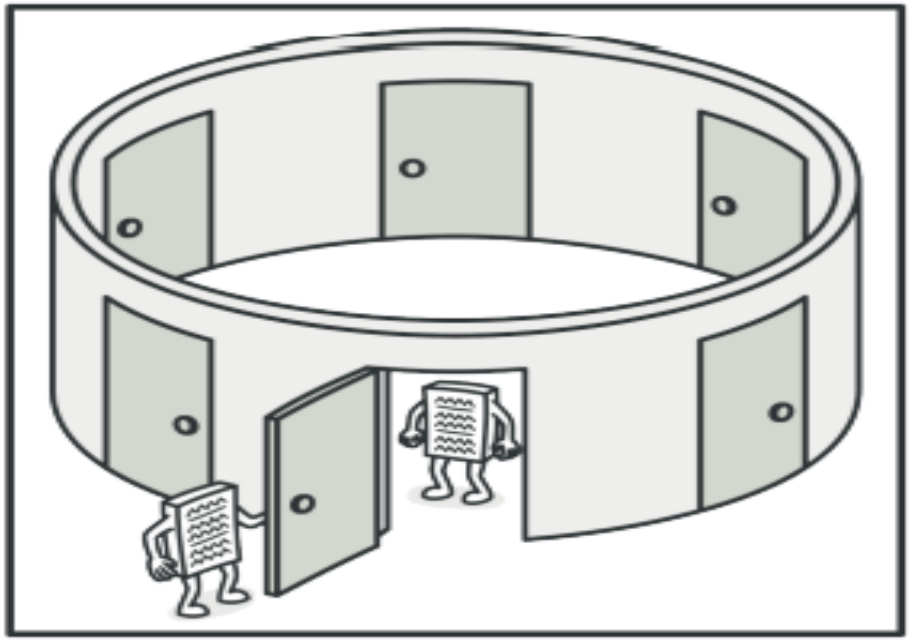
Creational Design Patterns

- **Singleton**
- **Factory Method**
- **Abstract Factory**
- **Builder**
- **Prototype**

Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance





Clients may not even realize that they're working with the same object all the time.

Singleton

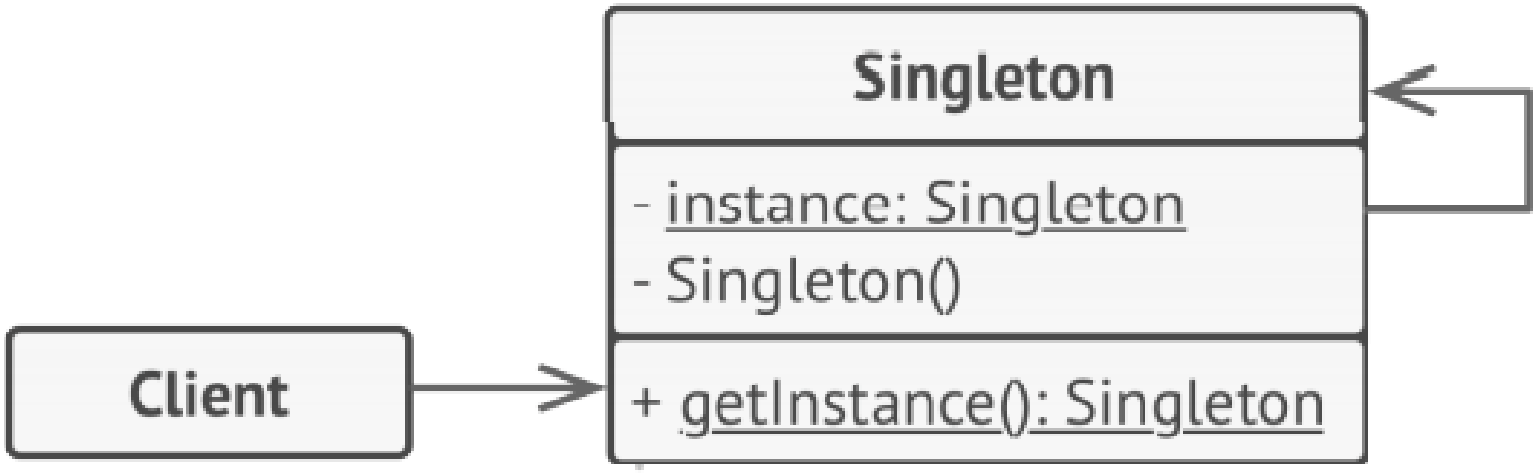
This behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design

Make the default constructor private

Create a static creation method that acts as a constructor

Singleton

Structure




```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

Singleton

Applicability

When a class in your program should have just a single instance available to all clients

When you need stricter control over global variables

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created

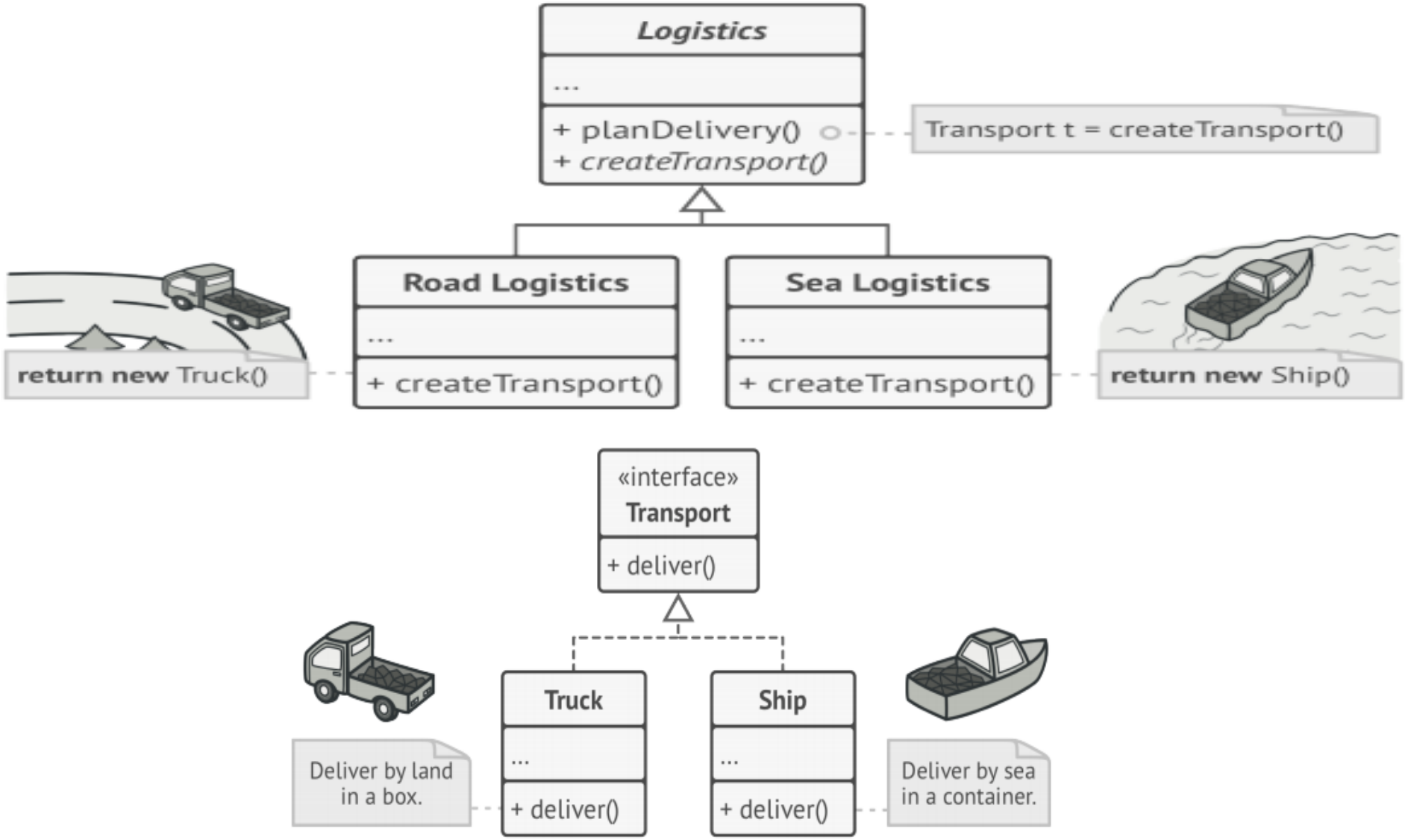
Problem

Factory Method

Replace direct object construction calls with calls to a special factory method

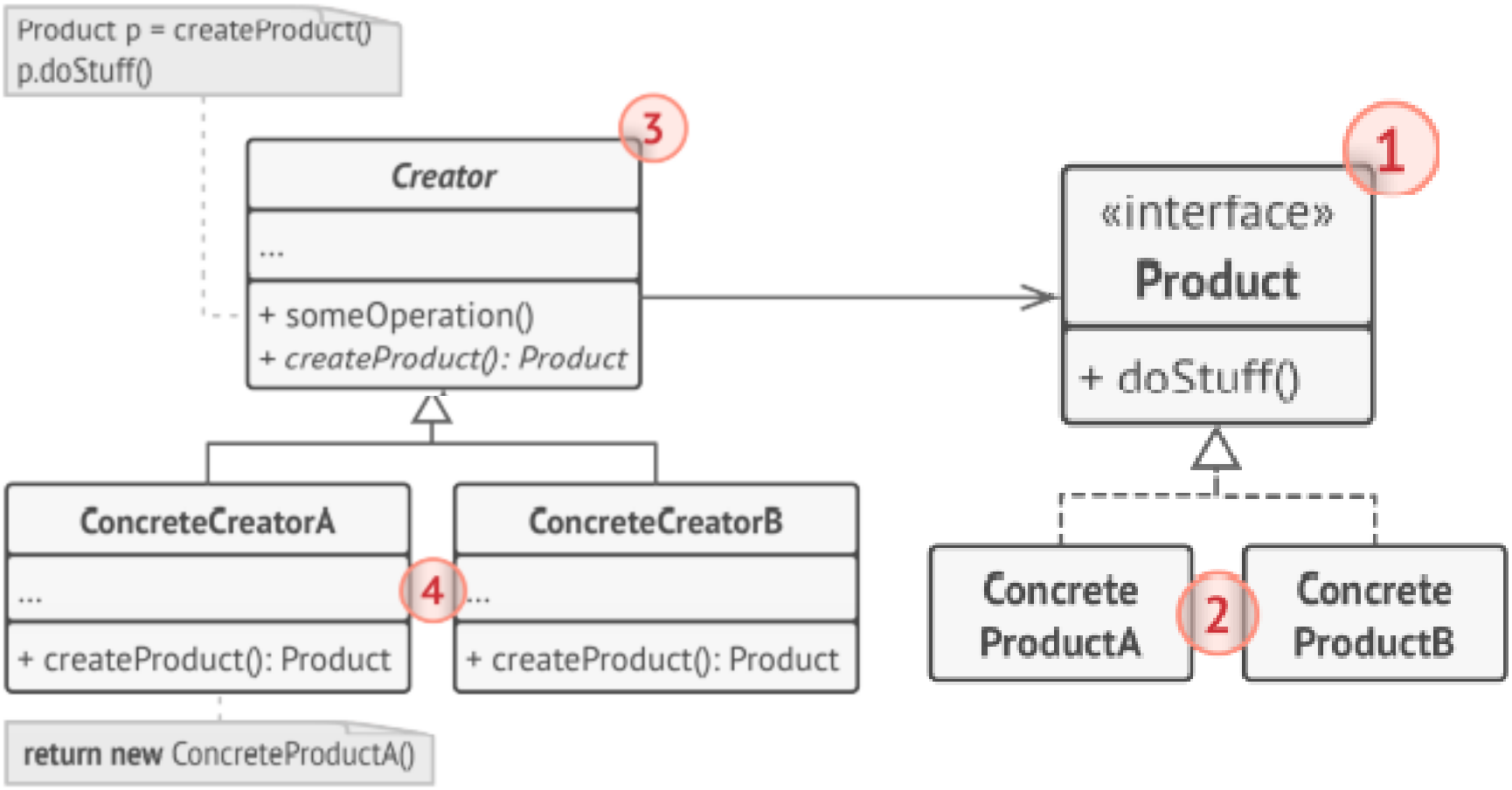
Objects returned by a factory method are often referred to as “products”

Factory Method



Factory Method

Structure



Factory Method

Applicability

- When you don’t know beforehand the exact types and dependencies of the objects your code should work with.

- When you want to provide users of your library or framework with a way to extend its internal components.

Factory Method

Pros & Cons

- Avoid tight coupling between the creator and the concrete products.
- Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support
- Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code
- The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

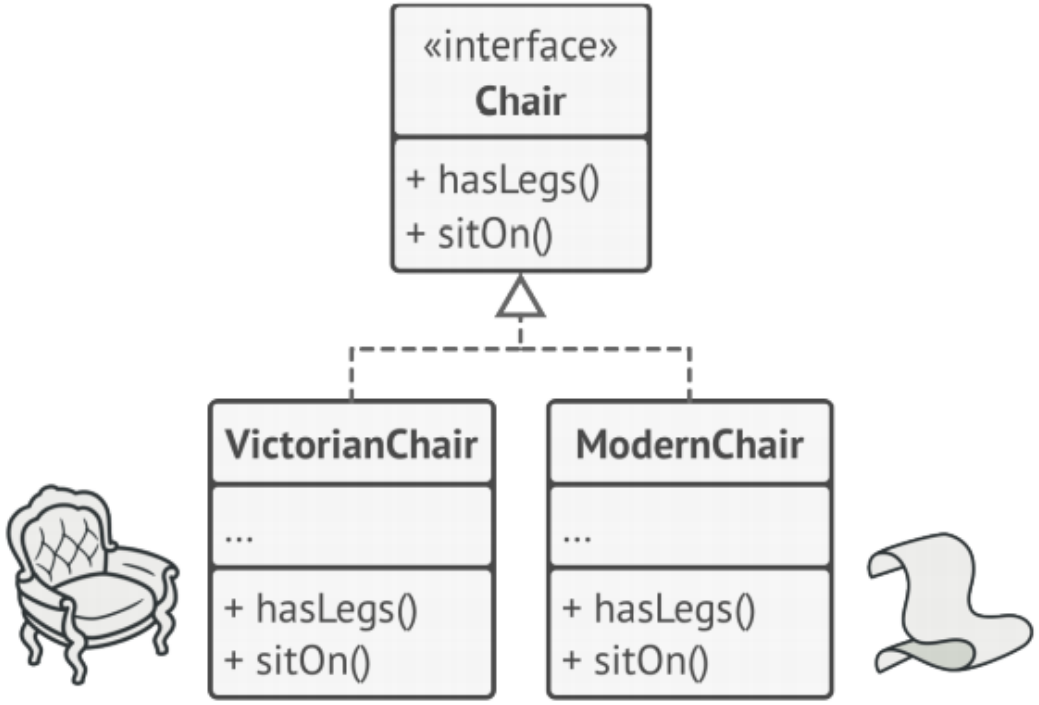
Problem

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

Abstract Factory

Declare interfaces for each distinct product of the product family (e.g., chair, sofa or coffee table)

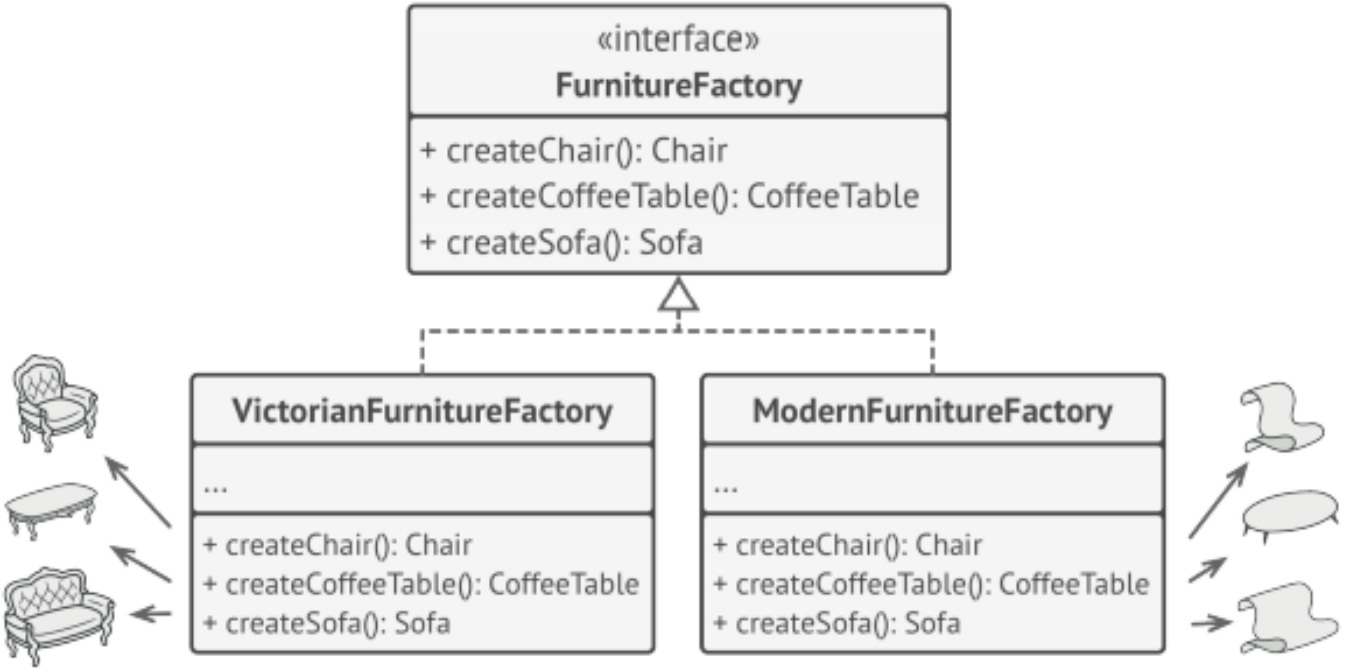
Then you can make all variants of products follow those interfaces



Abstract Factory

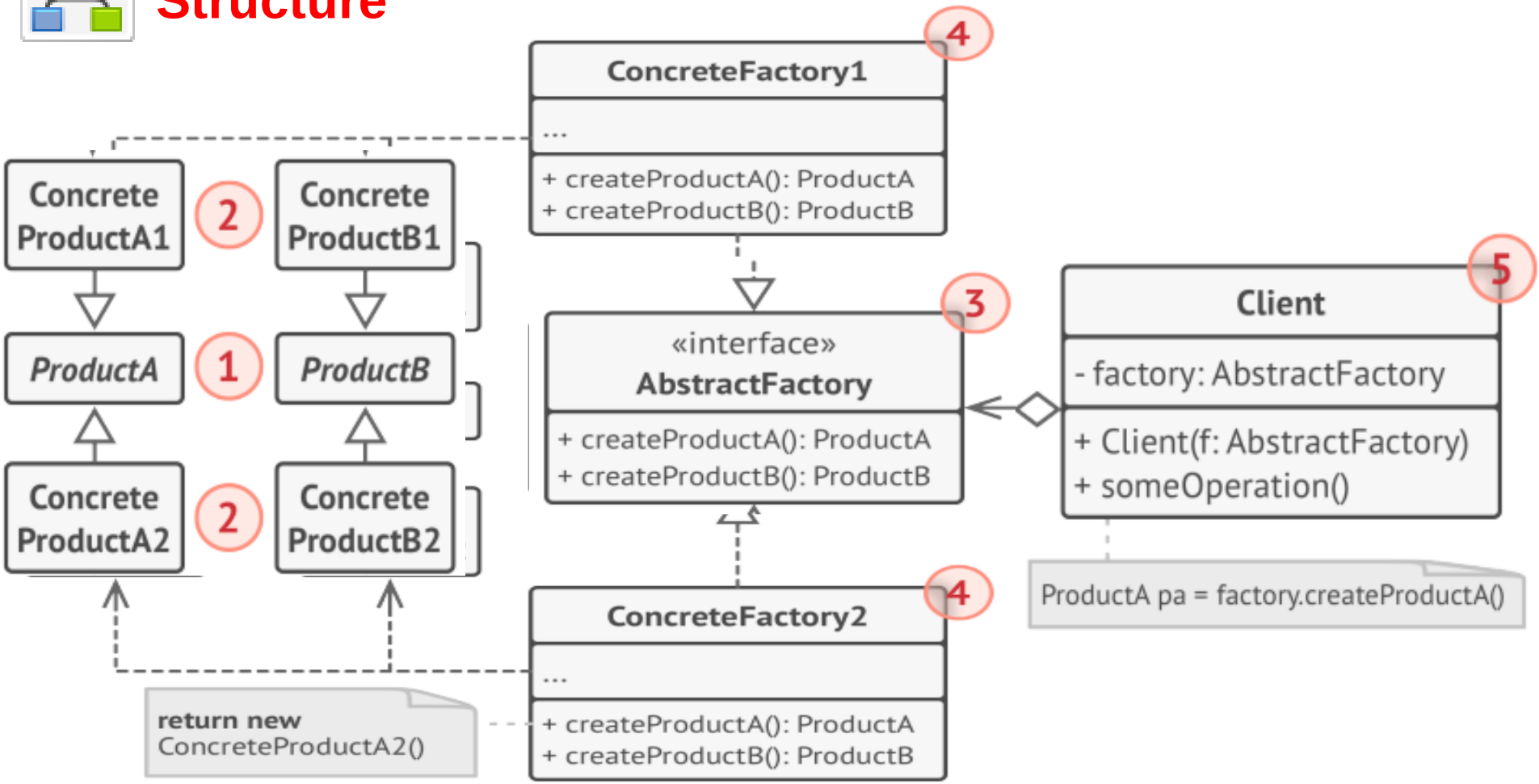
Declare the AbstractFactory interface with a list of creation methods for all products that are part of the product family

For each variant of a product family, we create a separate factory class based on the AbstractFactory interface



Abstract Factory

Structure



Abstract Factory

Applicability

When your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products

Abstract Factory

Pros & Cons

- You can be sure that the products you're getting from a factory are compatible with each other.
- You avoid tight coupling between concrete products and client code.
- Single Responsibility Principle. You can extract the product creation code into one place, making the code easier to support.
- Open/Closed Principle. You can introduce new variants of products without breaking existing client code.
- The code may become more complicated than it should be

Creational Design Patterns

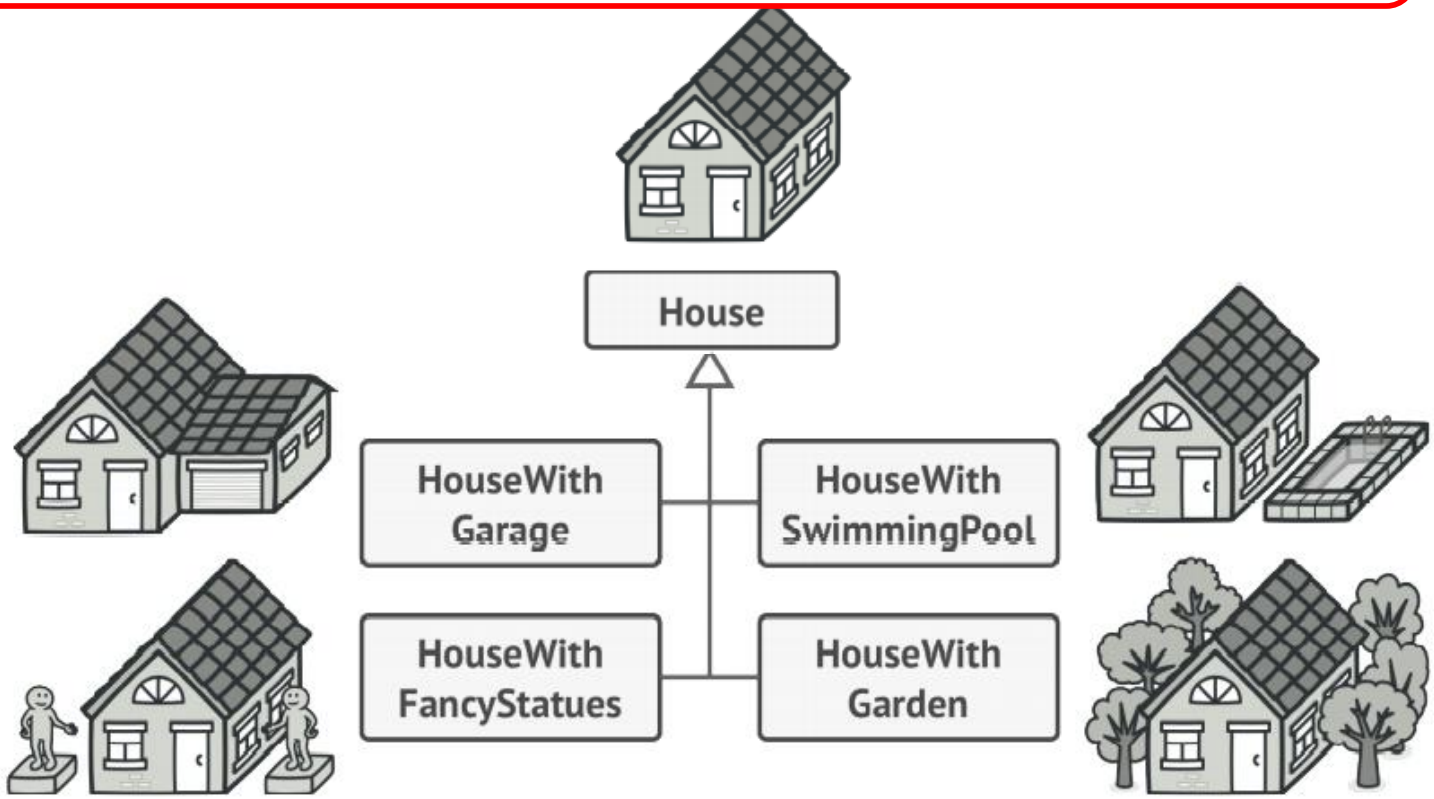
- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Builder

Lets you construct complex objects step by step

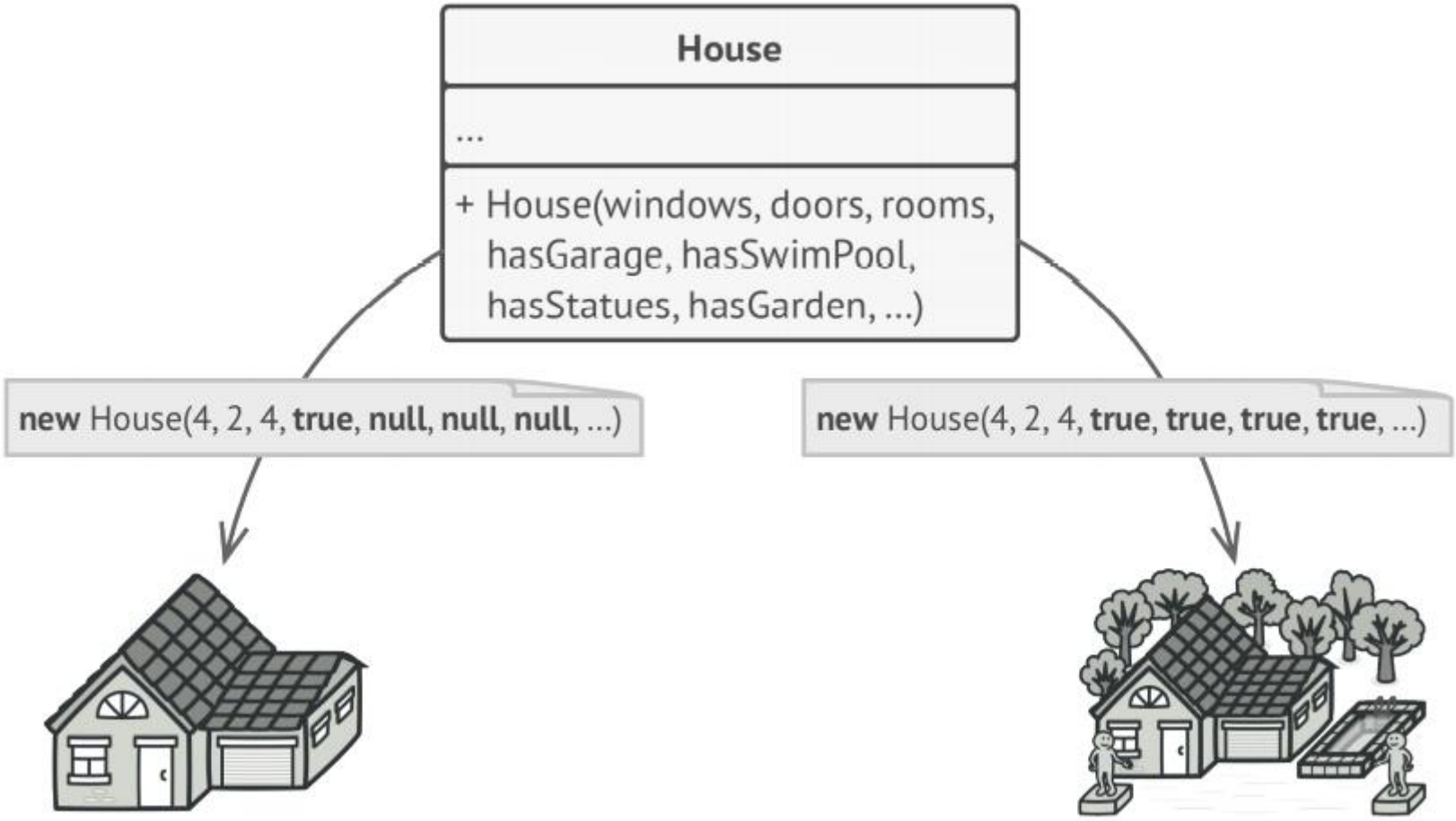
Produce different types and representations of an object using the same construction code

Problem



Builder

Problem

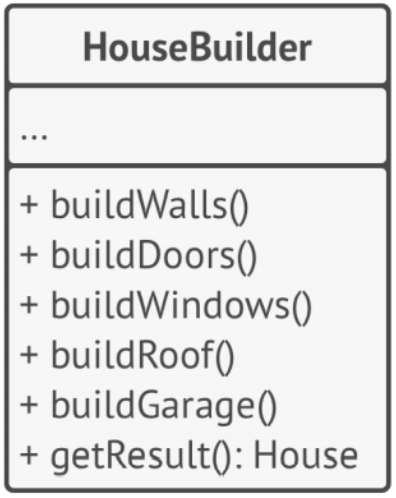


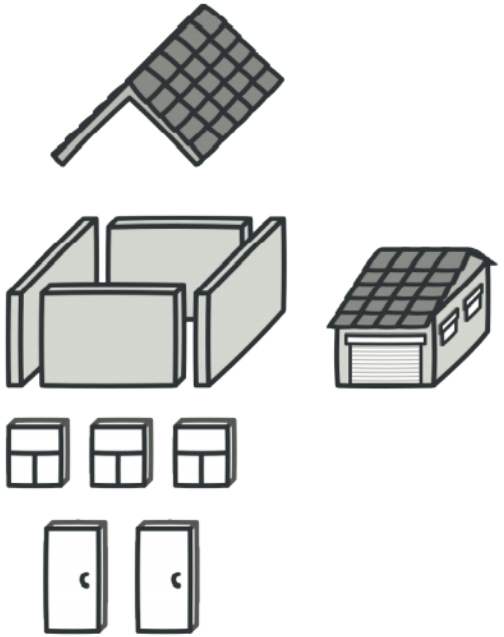
Builder

Extract the object construction code out of its own class and move it to separate objects called builders.

To create an object, you execute a series of these steps on a builder object

You don’t need to call all of the steps

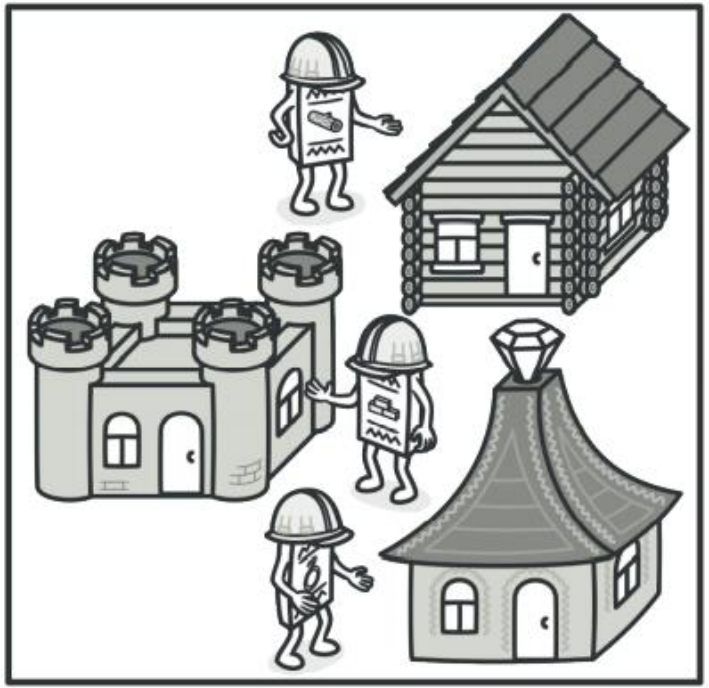




Builder

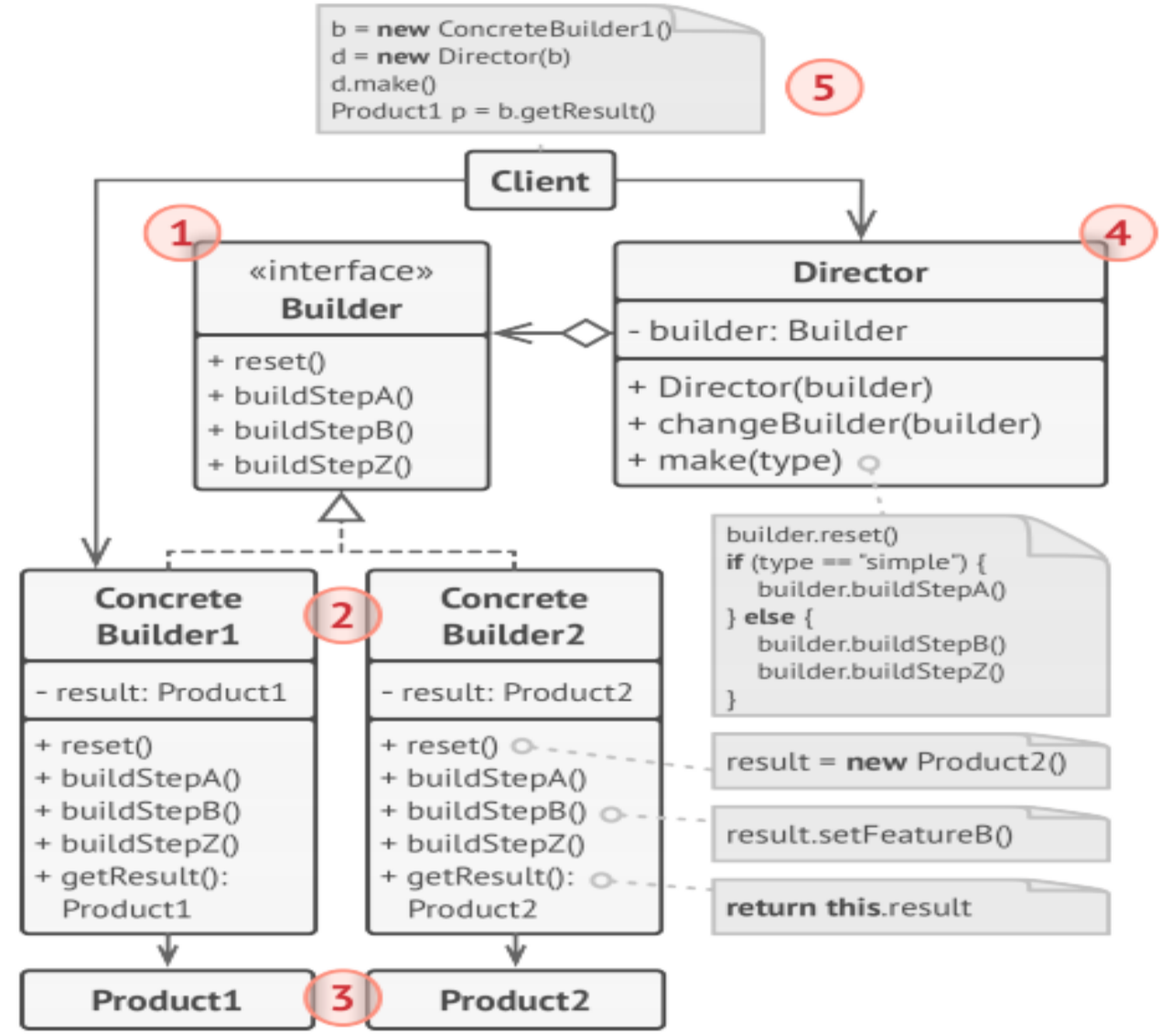
Walls of a cabin may be built of wood, but the castle walls must be built with stone

Create several different builder classes that implement the same set of building steps, but in a different manner



Builder

Structure



Builder

Applicability

To get rid of a “telescopic constructor”.

When you want your code to be able to create different representations of some product (for example, stone and wooden houses).

Construct Composite trees or other complex objects

Builder

Pros & Cons

- Construct objects step-by-step, defer construction steps or run steps recursively
- Reuse the same construction code when building various representations of products
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.
- The overall complexity of the code increases since the pattern requires creating multiple new classes

Creational Design Patterns

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype (Remember ICloneable !!)

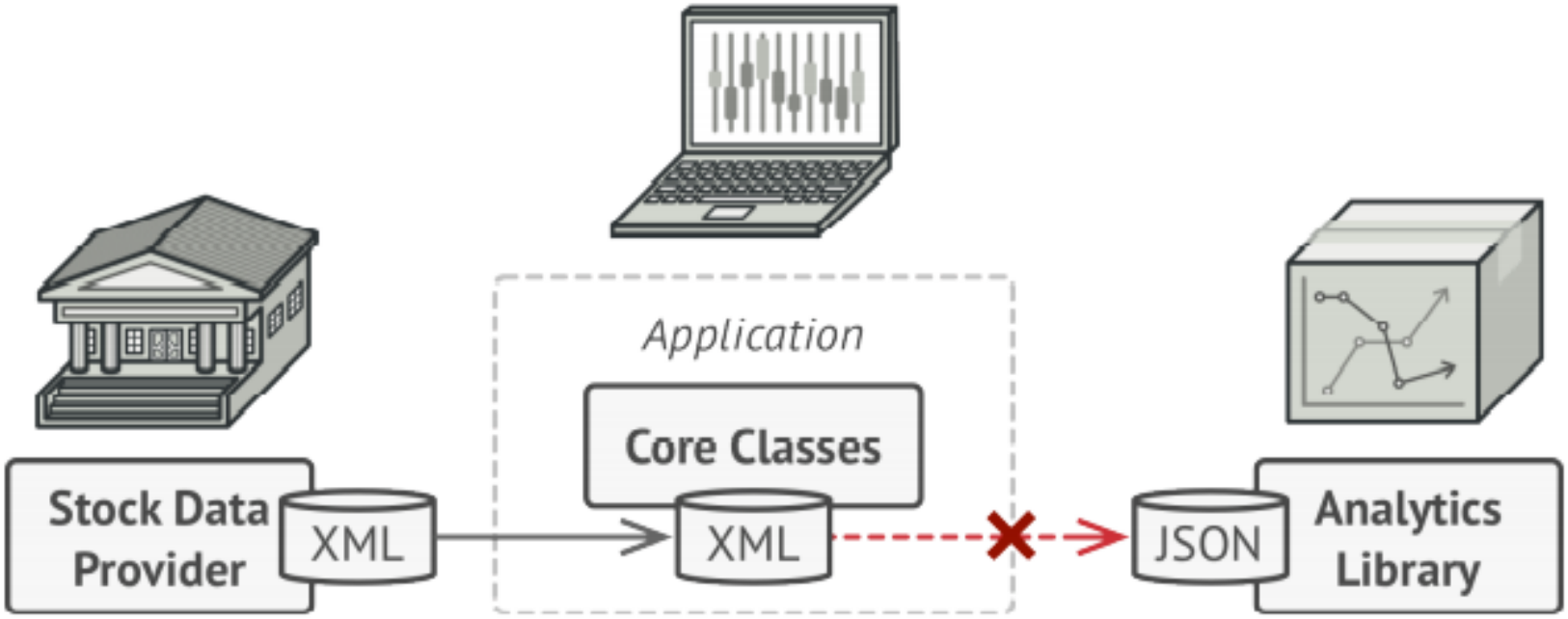
Structural Design Patterns

- **Adapter**
- **Bridge**
- **Composite**
- **Decorator**
- **Facade**
- **Flyweight**
- **Proxy**

Adapter

Allows objects with incompatible interfaces to collaborate.

Problem



Adapter

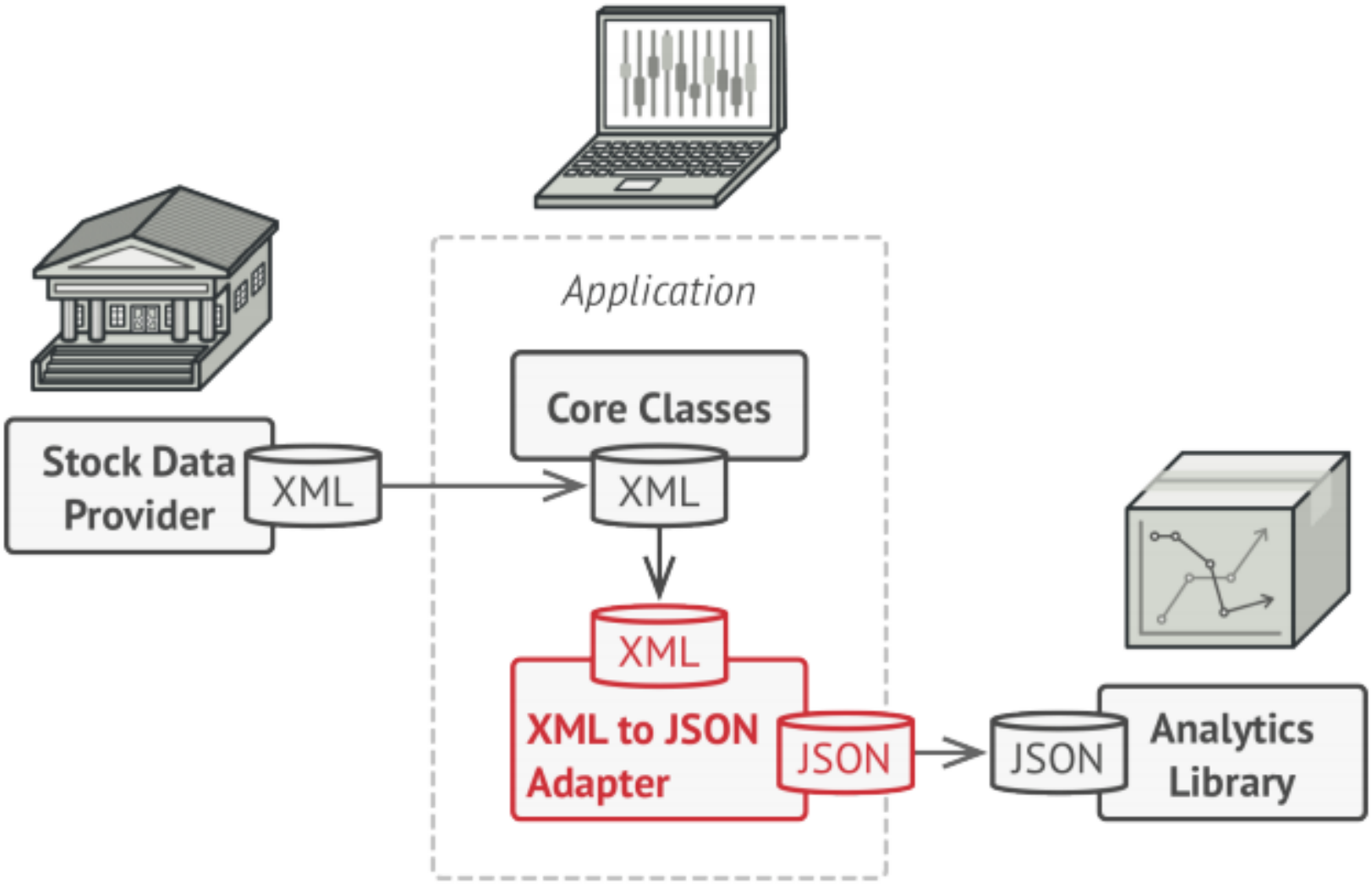
Create an adapter

Special object that converts the interface of one object so that another object can understand it

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes.

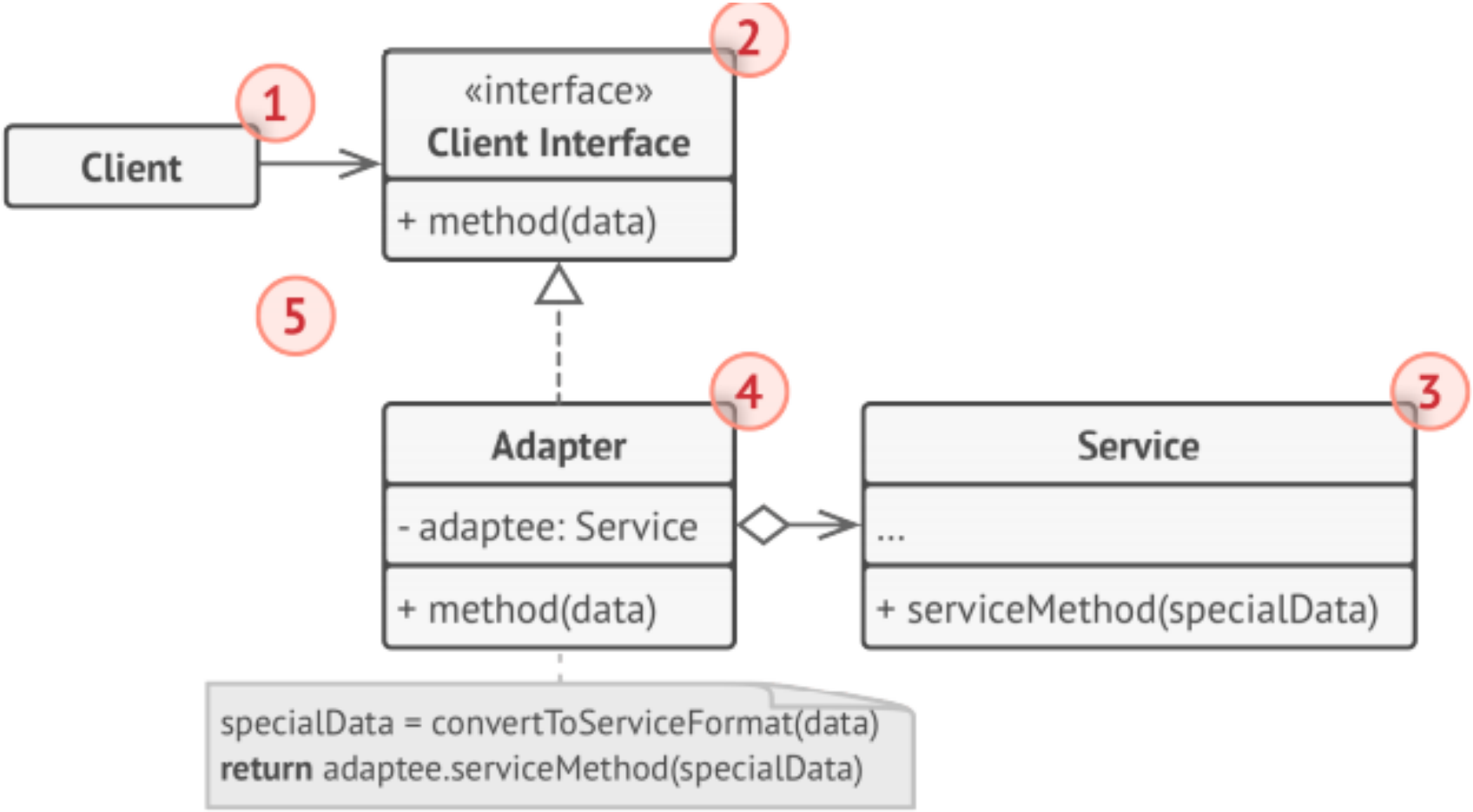
The wrapped object isn’t even aware of the adapter

Adapter


Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

Adapter

Structure



Adapter

Applicability

When you want to use some existing class, but its interface isn't compatible with the rest of your code

When you want to reuse several existing classes that lack some common functionality that can't be added to the superclass

Adapter

Pros & Cons

Single Responsibility Principle. You can separate the interface or data conversion code from the primary business logic of the program

Open/Closed Principle. You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface

The overall complexity of the code increases since the pattern requires creating multiple new interfaces & classes

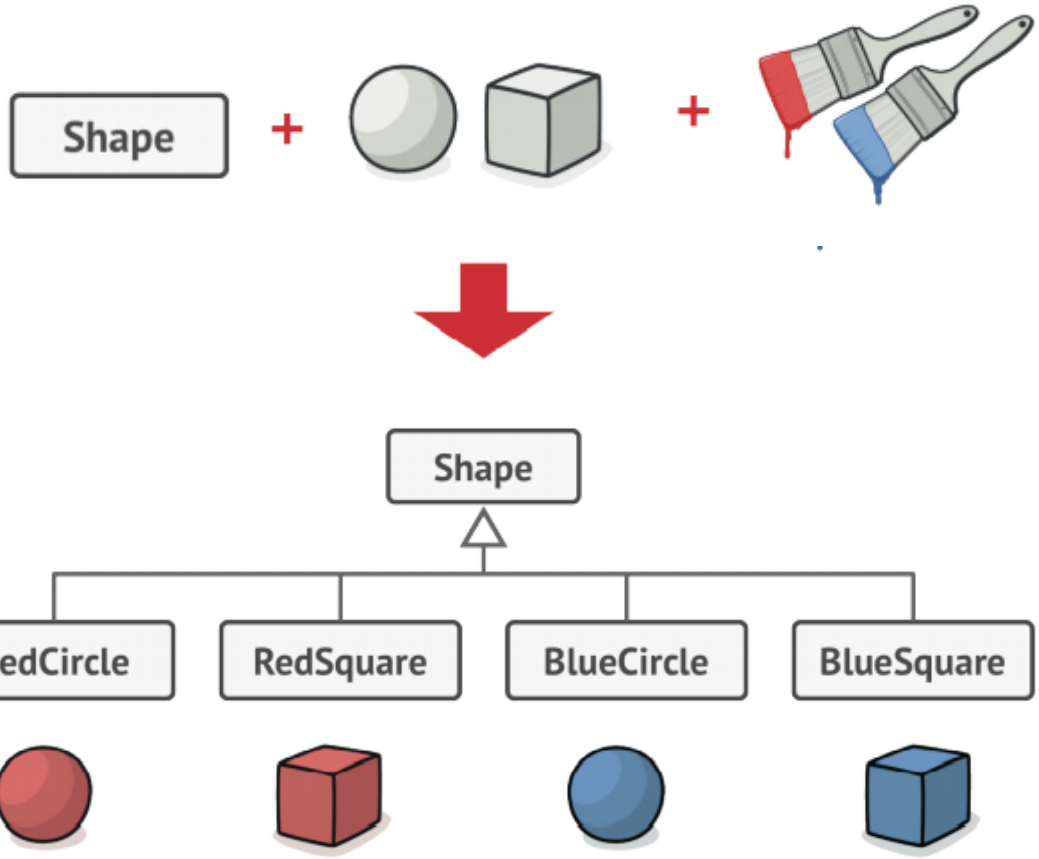
Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies

Problem



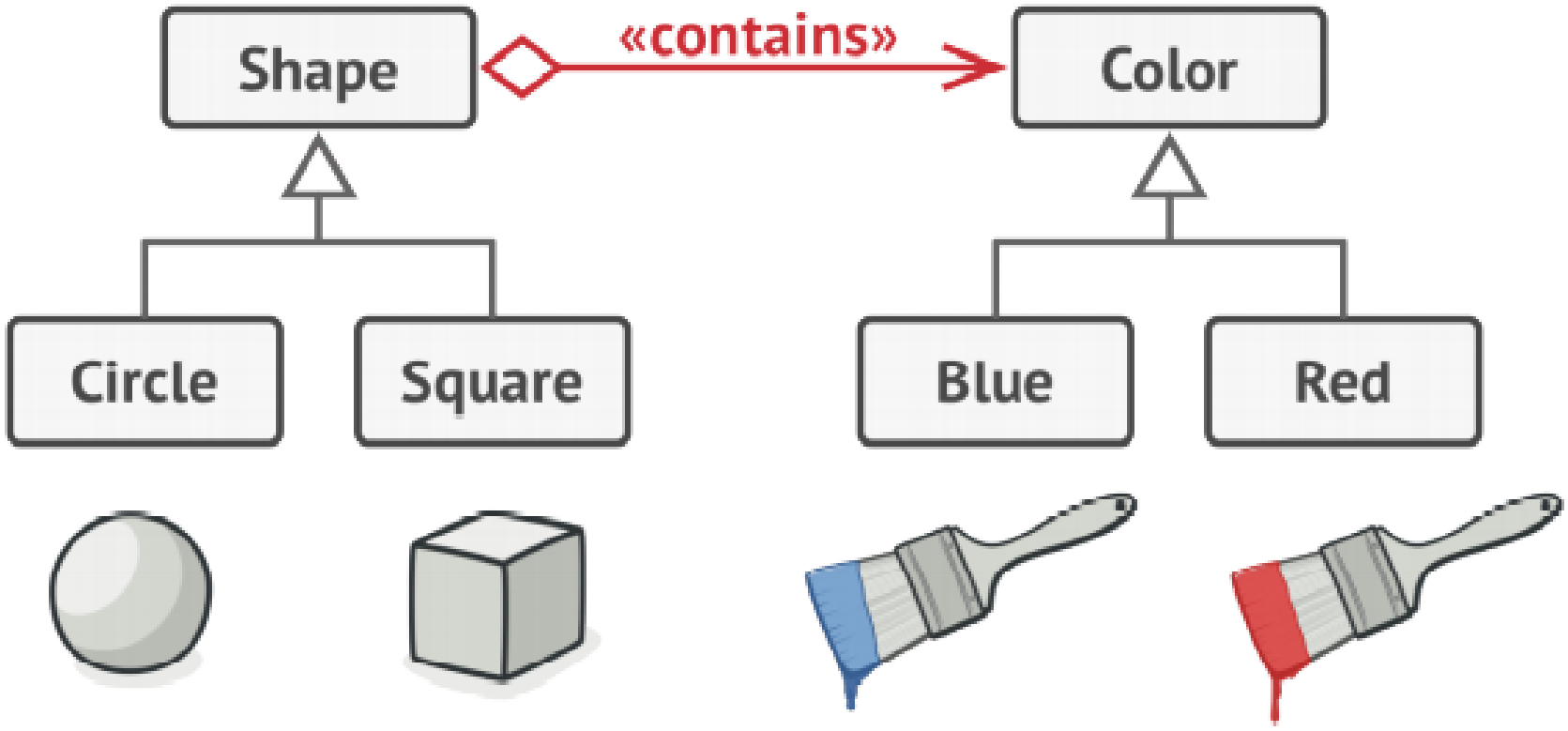
Bridge

This occurs because we're trying to extend the shape classes in two independent dimensions

Switch from inheritance to composition

Extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy

Bridge



Bridge

Applicability

- When you want to divide and organize a monolithic class that has several variants of some functionality

- When you need to extend a class in several orthogonal (independent) dimensions

Bridge

Pros & Cons

- Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation
- You can create platform-independent classes and apps **(**The client code works with high-level abstractions. It isn't exposed to the platform details.**)**
- The overall complexity of the code increases since the pattern requires creating multiple new interfaces & classes

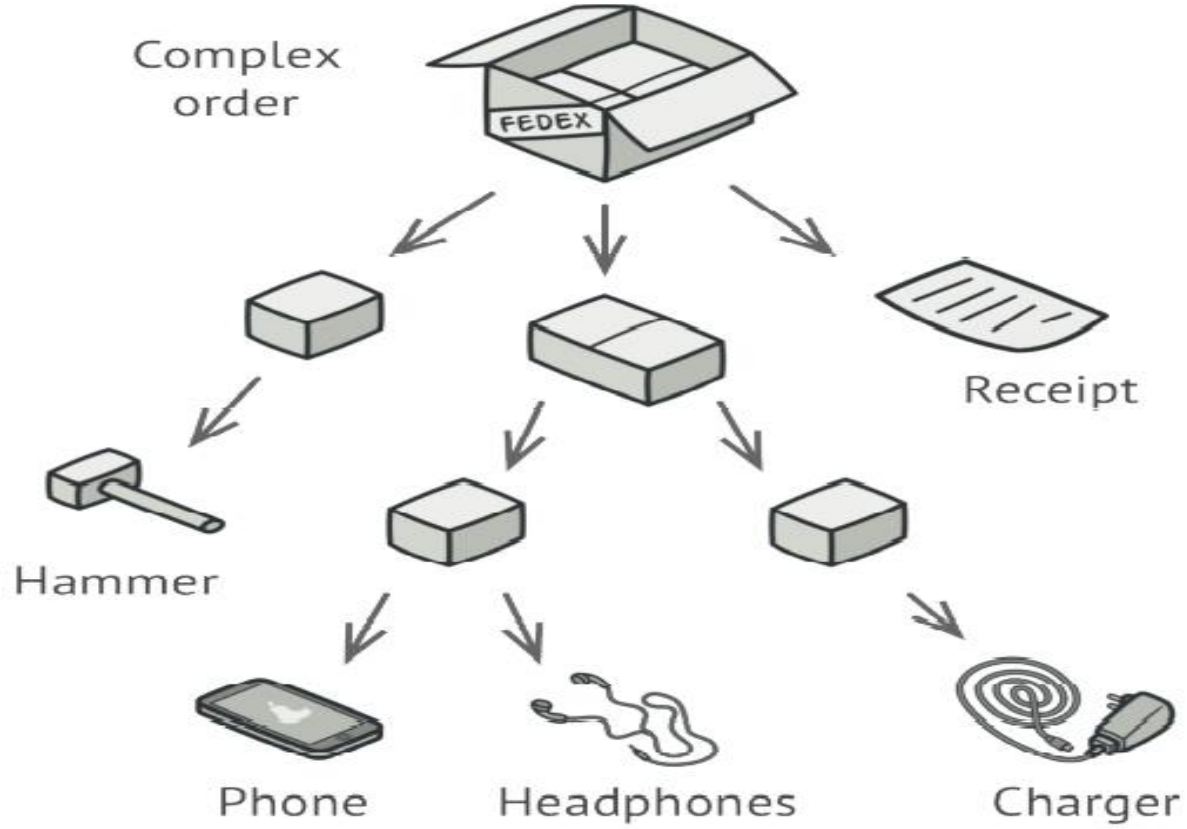
Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects

Problem



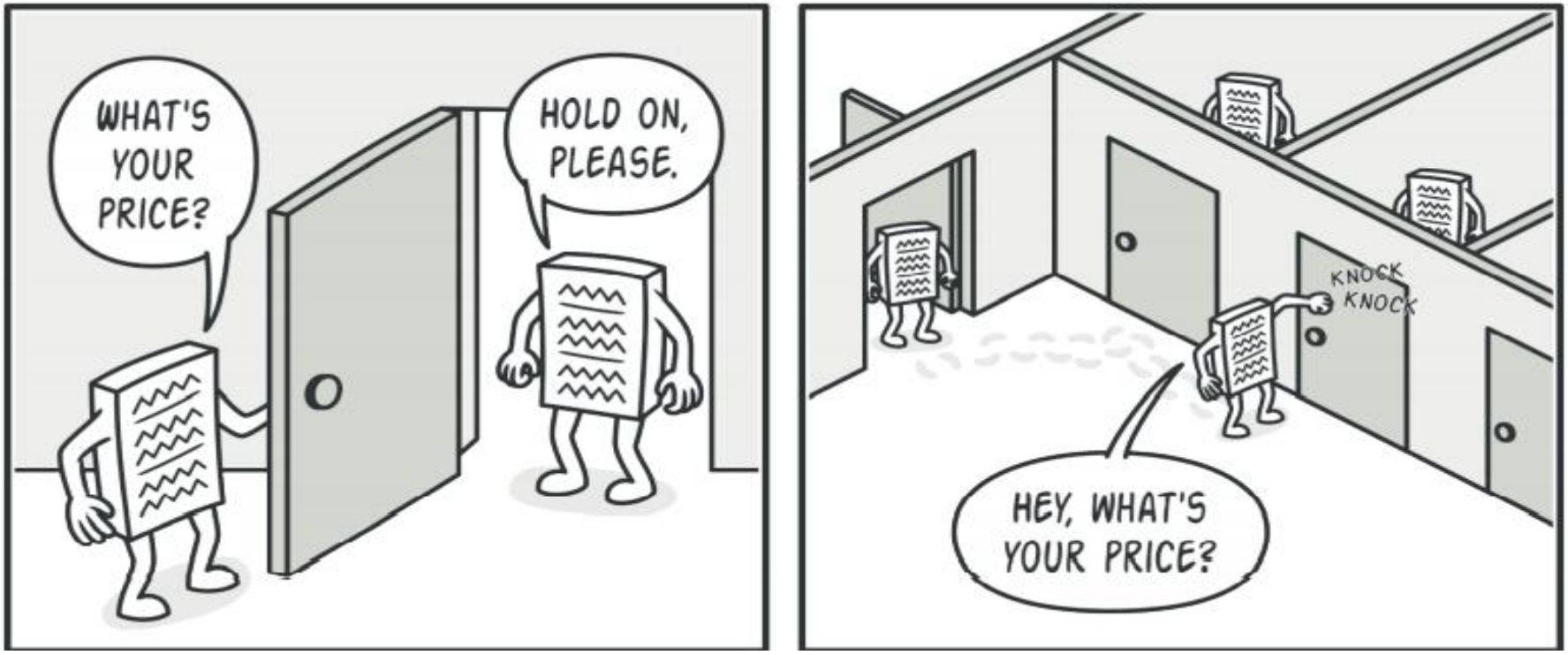
Composite

Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes.

How would you determine the total price of such an order?

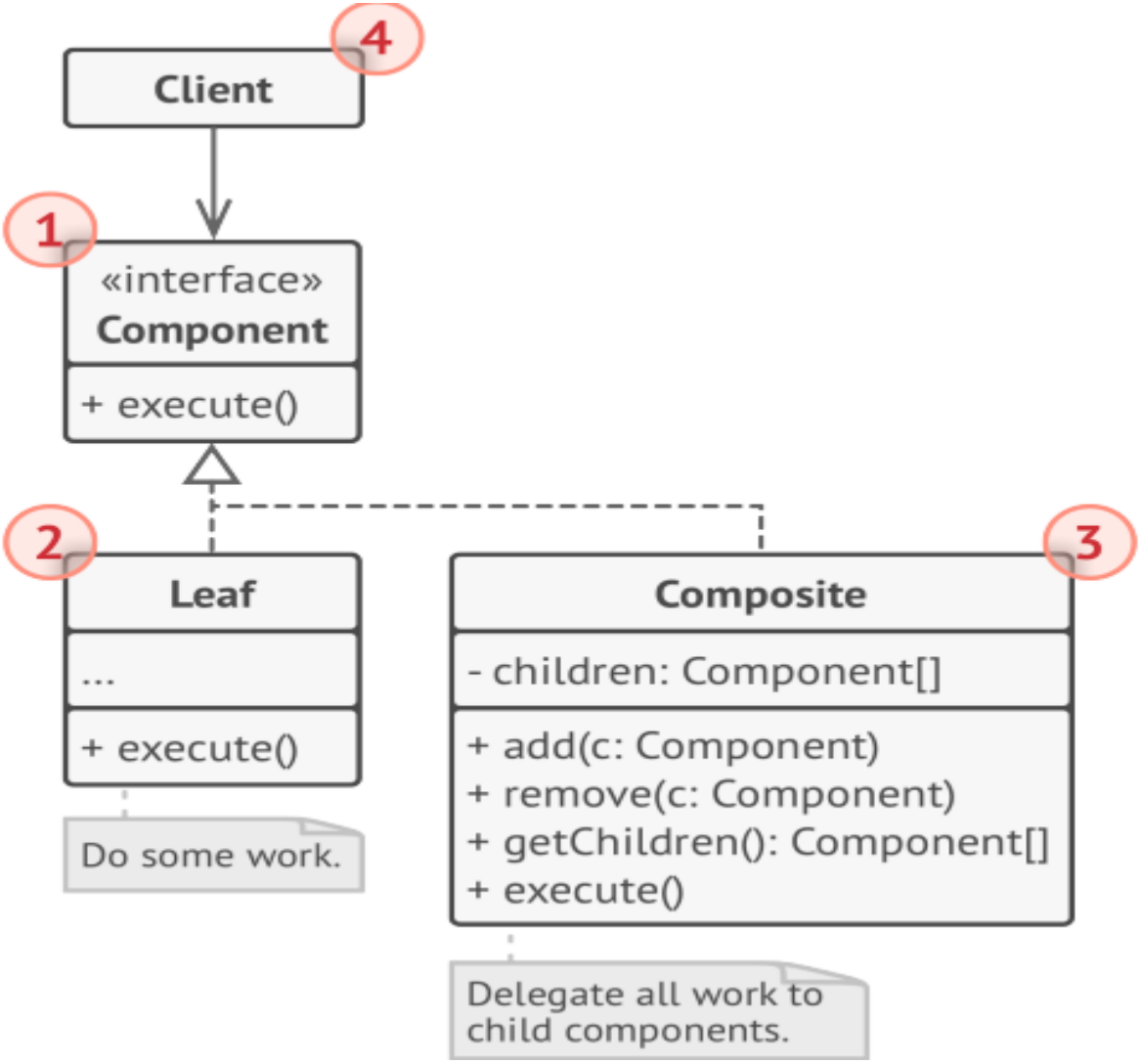
Composite

Work with Products and Boxes through a common interface which declares a method for calculating the total price.



Composite

Structure



Composite

Applicability

When you have to implement a tree-like object structure

When you want the client code to treat both simple and complex elements uniformly.

Composite

Pros & Cons

- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage

- Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree

- It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

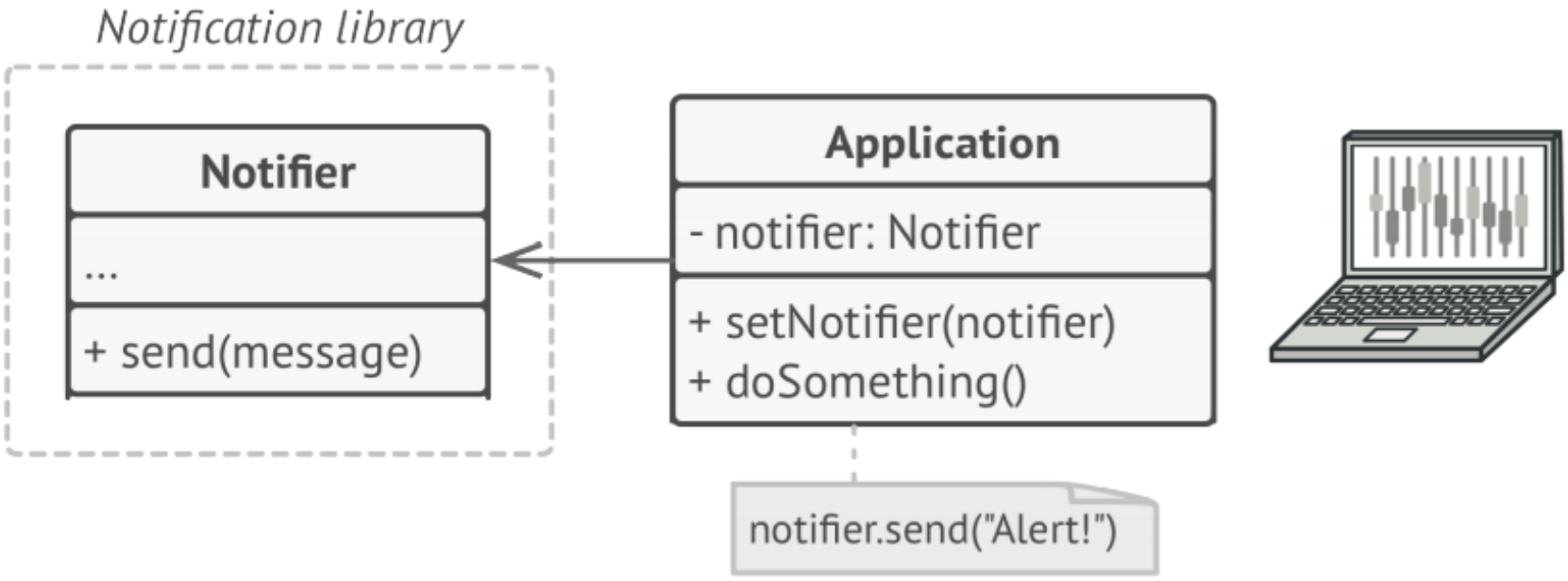
Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Decorator

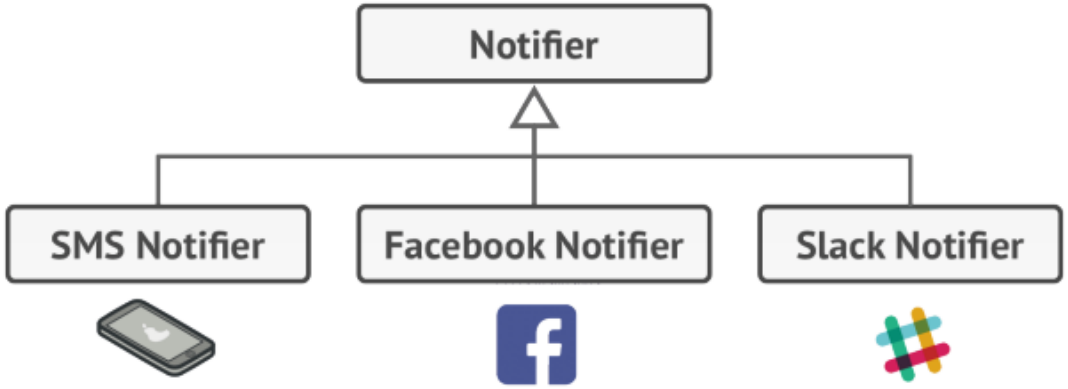
Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors

Problem

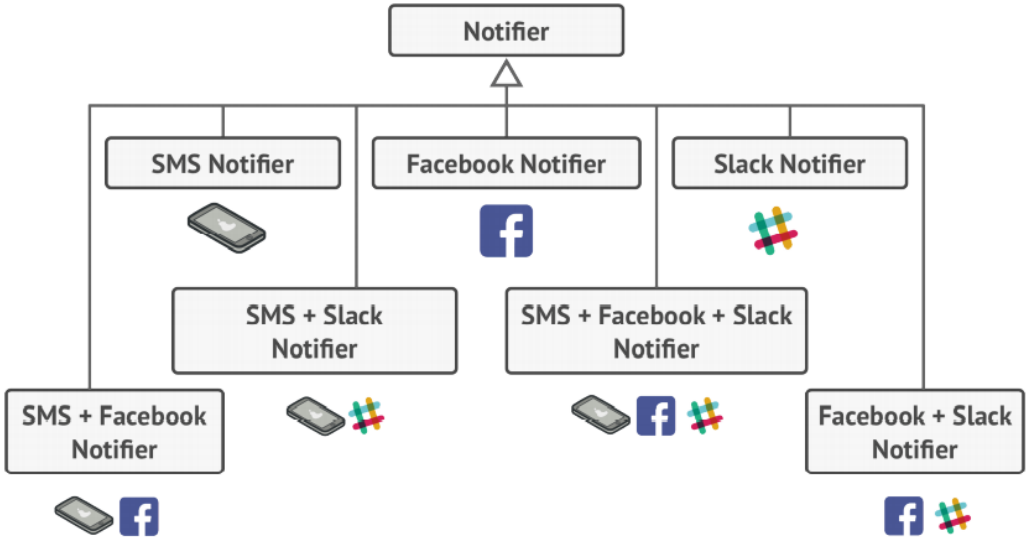


Decorator

I’m expecting more than just email notifications



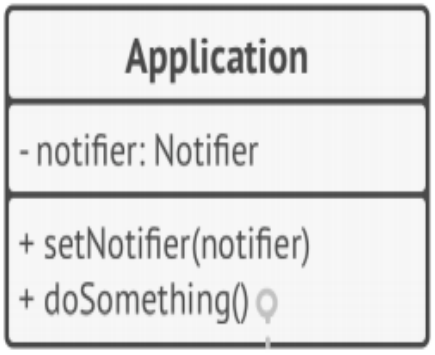
Why can’t you use several notification types at once

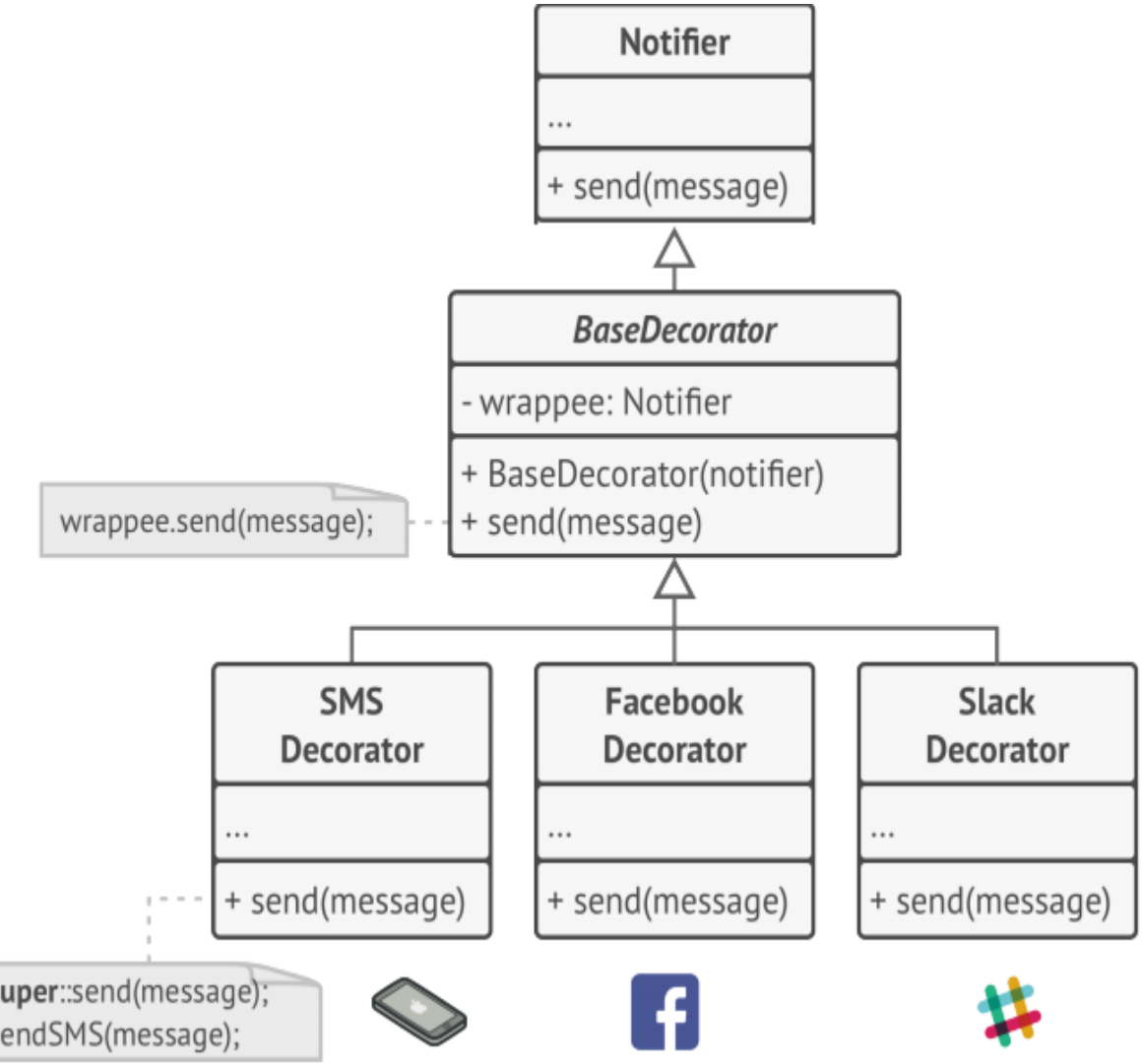


Decorator

Use Wrappers

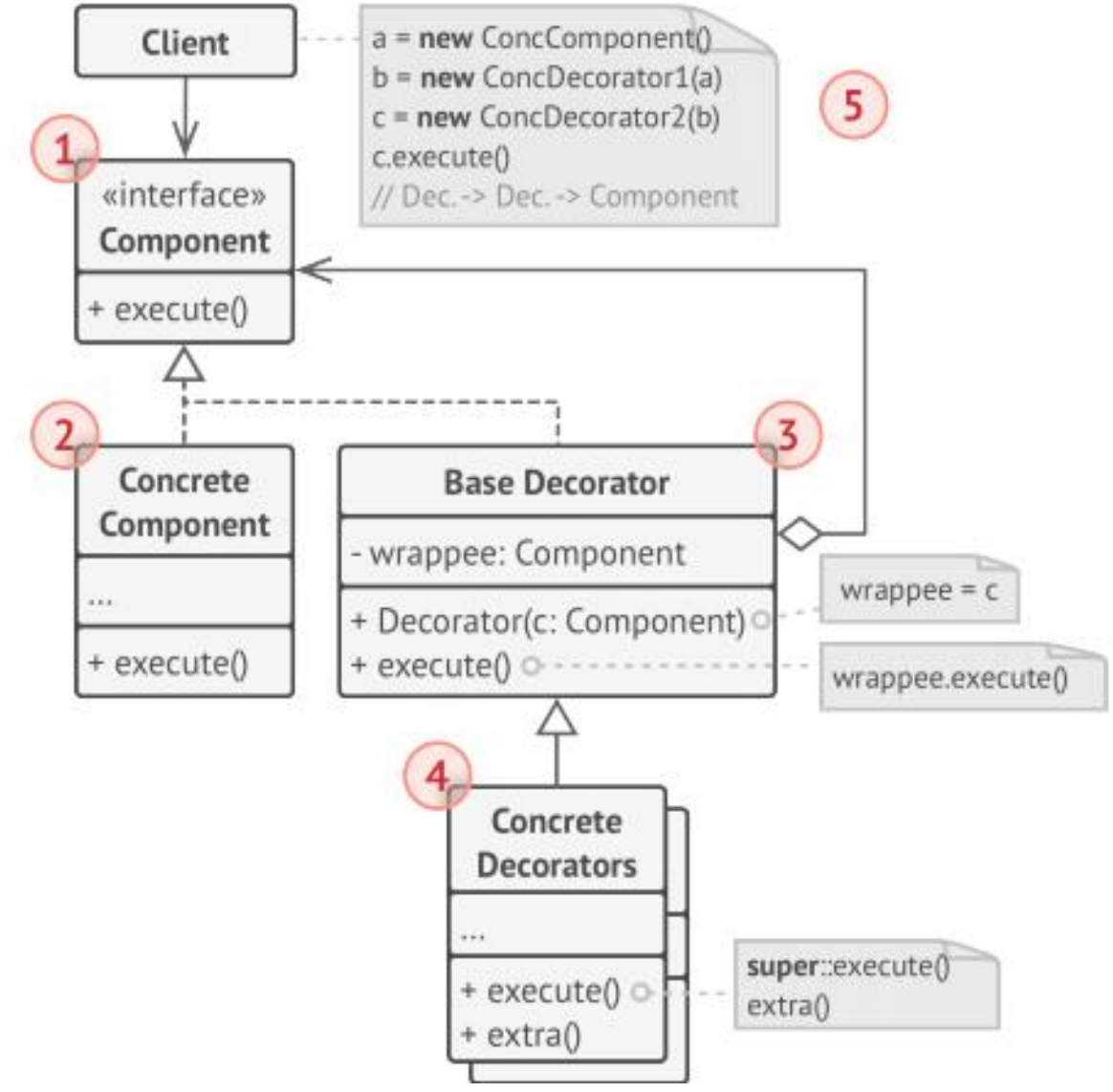
```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```





Decorator

Structure



Decorator

Applicability

When you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects

When it’s awkward or not possible to extend an object’s behavior using inheritance

Decorator

Pros & Cons

- You can extend an object's behavior without making a new subclass
- You can add or remove responsibilities from an object at runtime
- It's hard to remove a specific wrapper from the wrappers stack
- It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack

Thank You…