

# Software Design Patterns

---

## *Lecture 2*

## ***Software Design Principles***

**Dr. Fan Hongfei**  
**12 September 2024**

# Feature of Good Design (1)

---

- **Code reuse**

- **Challenge:** tight coupling between components, dependencies on concrete classes instead of interfaces, hardcoded operations
- **Solution:** design patterns
  - However, sometimes making components more complicated
- **Three levels of reuse:** a piece of wisdom from Erich Gamma
  - Lowest: classes
  - Highest: frameworks
  - **Middle level: design patterns**

# Feature of Good Design (2)

---

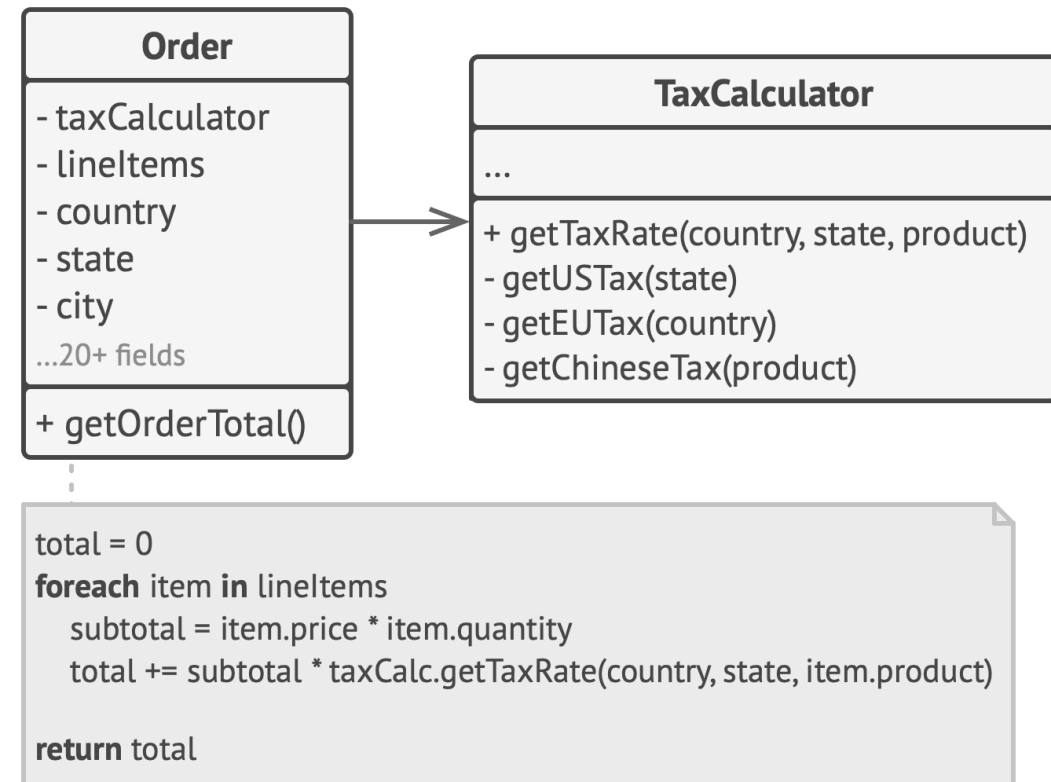
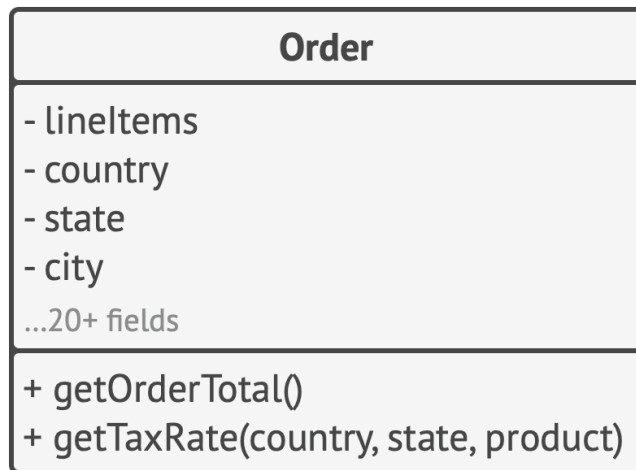
- **Extensibility**

- **Change** is the only constant thing in a programmer's life
  - We understand the problem better once we start to solve it
  - Something beyond your control has changed
  - Objectives/requirements have changed
- **Prepare for possible future changes** when designing an architecture

# Good Design Principles (1)

- **Encapsulate what varies**

- Identify the aspects of your application that vary, and separate them from what stays the same
- **Main goal: minimize the effect caused by changes**
- Isolating program parts that vary in independent modules, protecting the rest
- Encapsulation on a **method level**
- Encapsulation on a **class level**



# Good Design Principles (2)

---

- **Program to an interface, not an implementation**

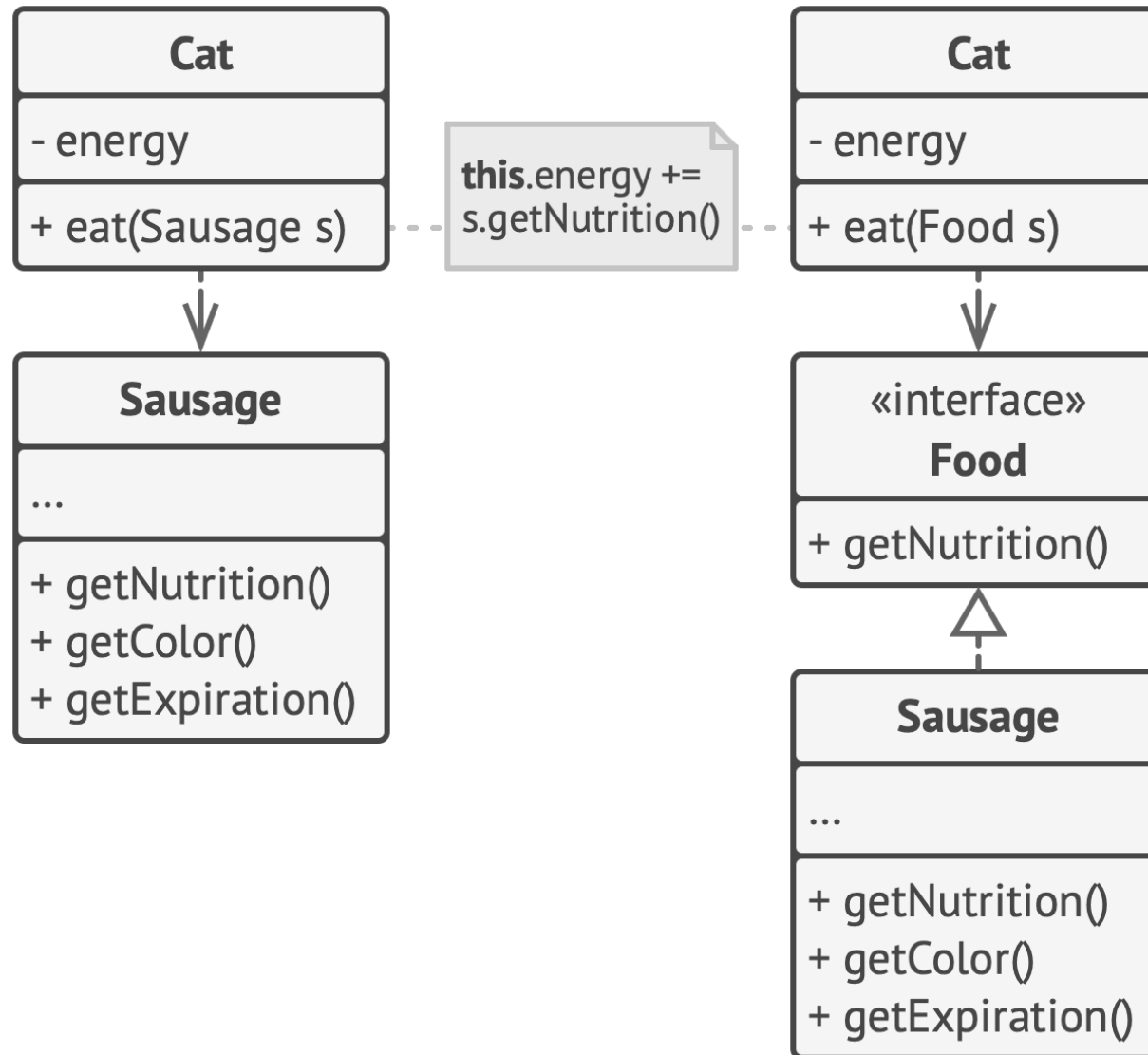
- In other words, **depend on abstractions**, not on concrete classes
- The design is flexible enough if you can easily extend it **without breaking existing code**

- **A possible approach**

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

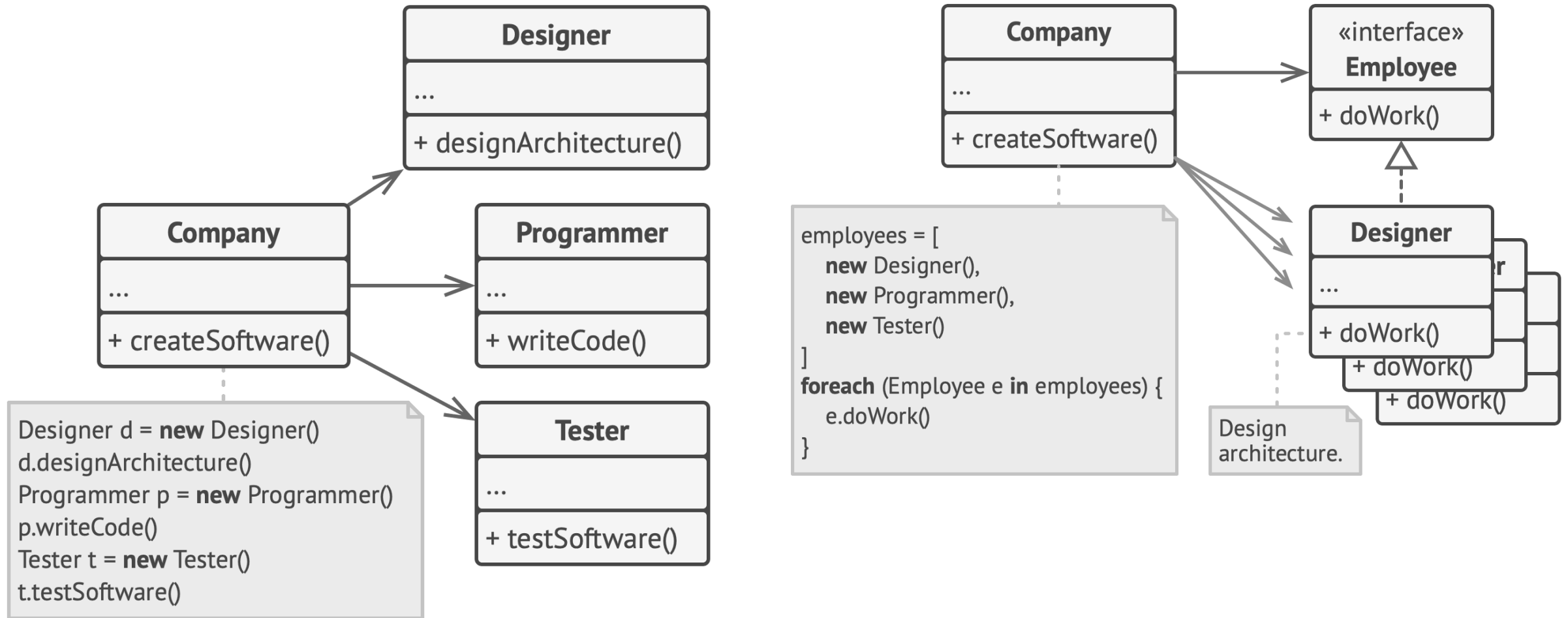
# Good Design Principles (2) (cont.)

- **Example 1**



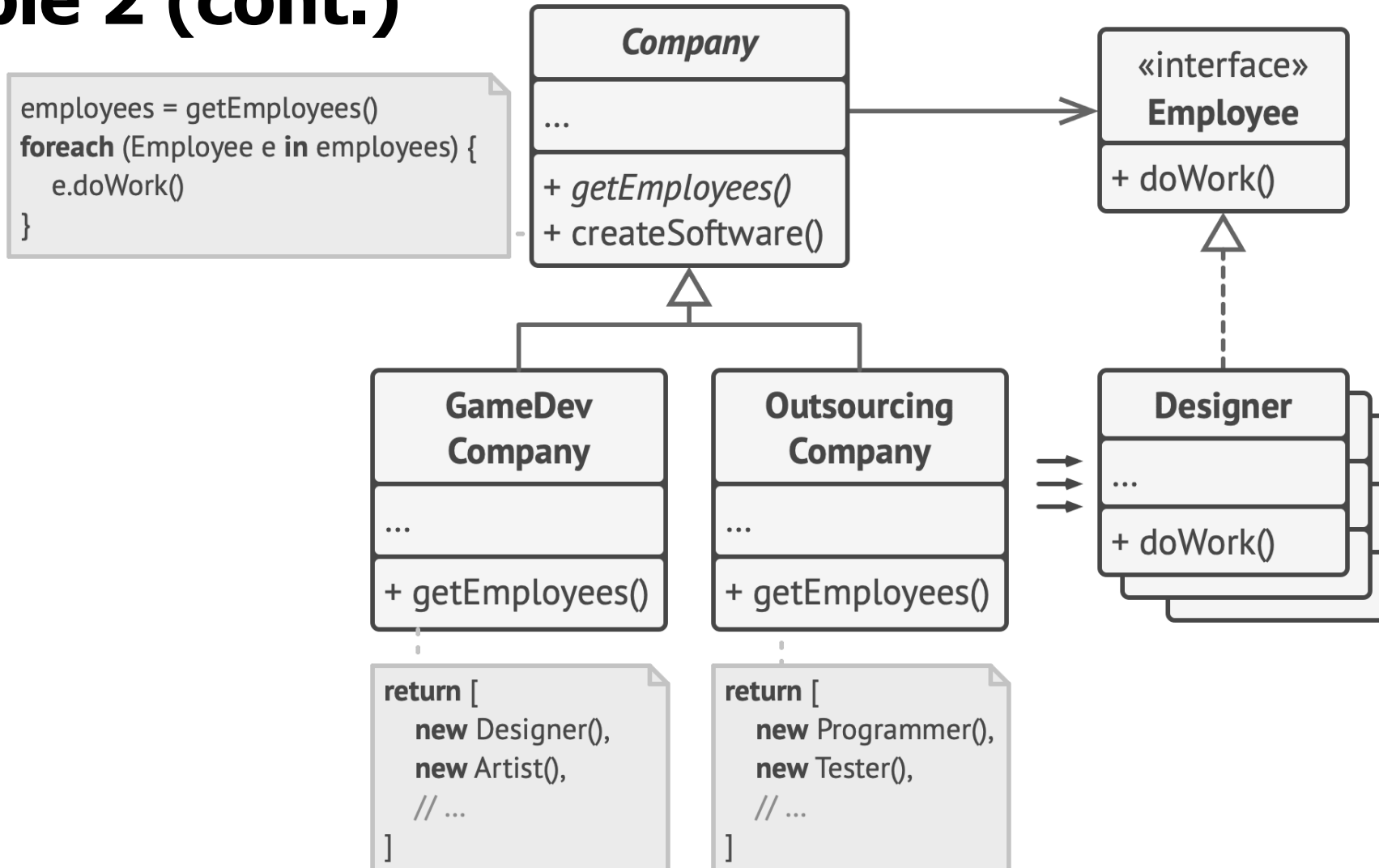
# Good Design Principles (2) (cont.)

## • Example 2



# Good Design Principles (2) (cont.)

- **Example 2 (cont.)**





# Good Design Principles (3)

---

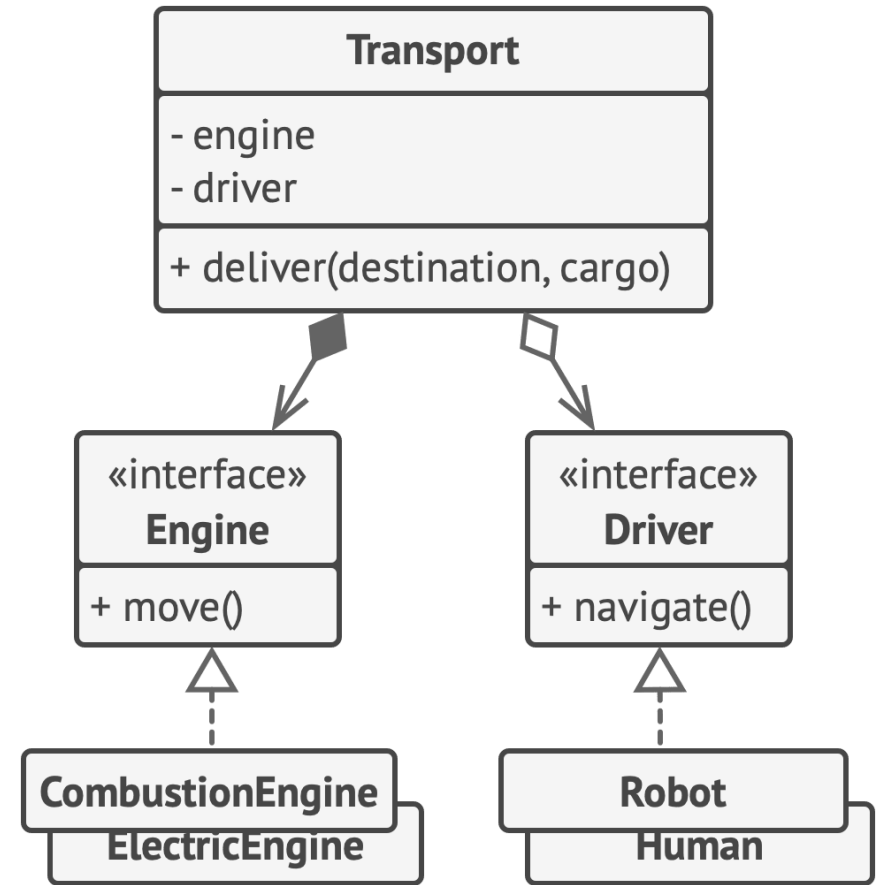
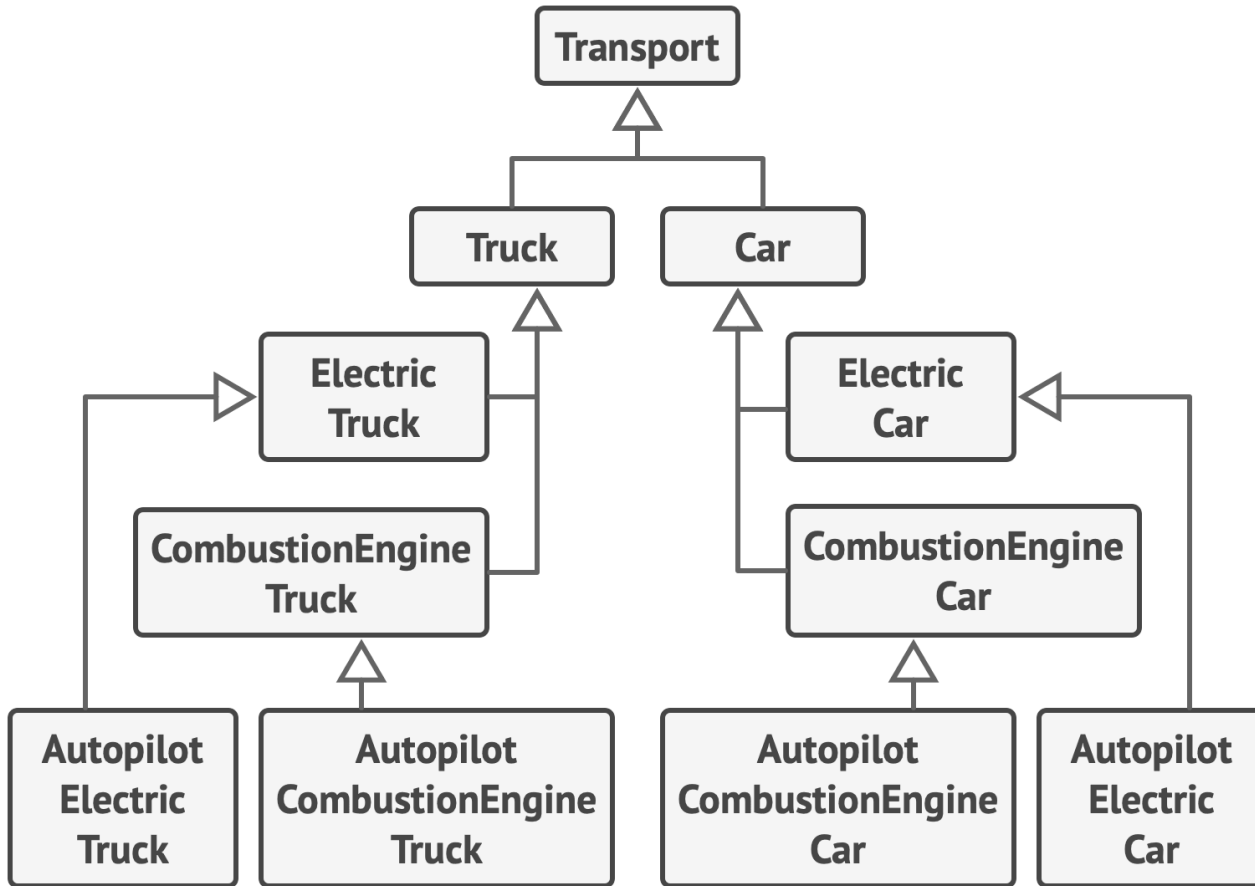
- **Favor composition over inheritance**

- **Challenge of inheritance**

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

# Good Design Principles (3) (cont.)

- **Inheritance vs. composition: “is a” vs. “has a”**



# SOLID Principles

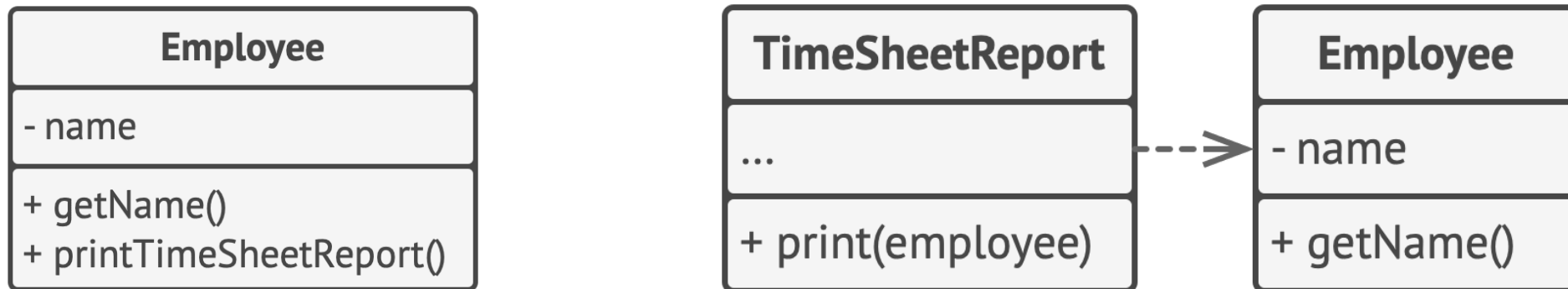
---

- **Agile Software Development: Principles, Patterns, and Practices**, by Robert Martin
- **Five design principles** intended to **make software designs more understandable, flexible and maintainable**
- However, using these principles mindlessly can cause more harm than good
  - **Always be pragmatic**

# SOLID 1: Single Responsibility Principle

---

- A class should have **just one reason to change**
  - Try to make every class responsible for a single part of functionality, and make that responsibility entirely encapsulated
- Main goal: reducing complexity, and reducing risks



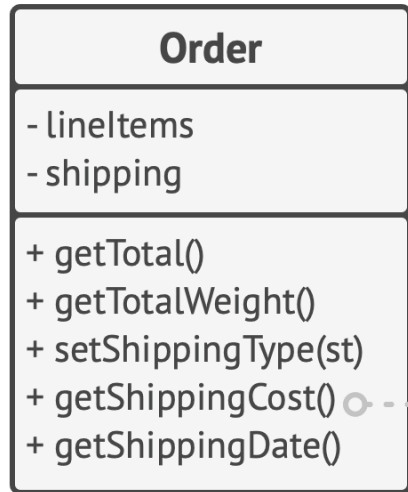
# SOLID 2: Open/Closed Principle

---

- Classes should be **open for extension** but **closed for modification**
  - Keep existing code from breaking when implementing new features
- A class is **open** if it can be extended by subclasses
- A class is **closed** if it is 100% ready to be used by others
- A class can be both open (for extension) and closed (for modification) at the same time
  - Not necessary to be applied for all changes

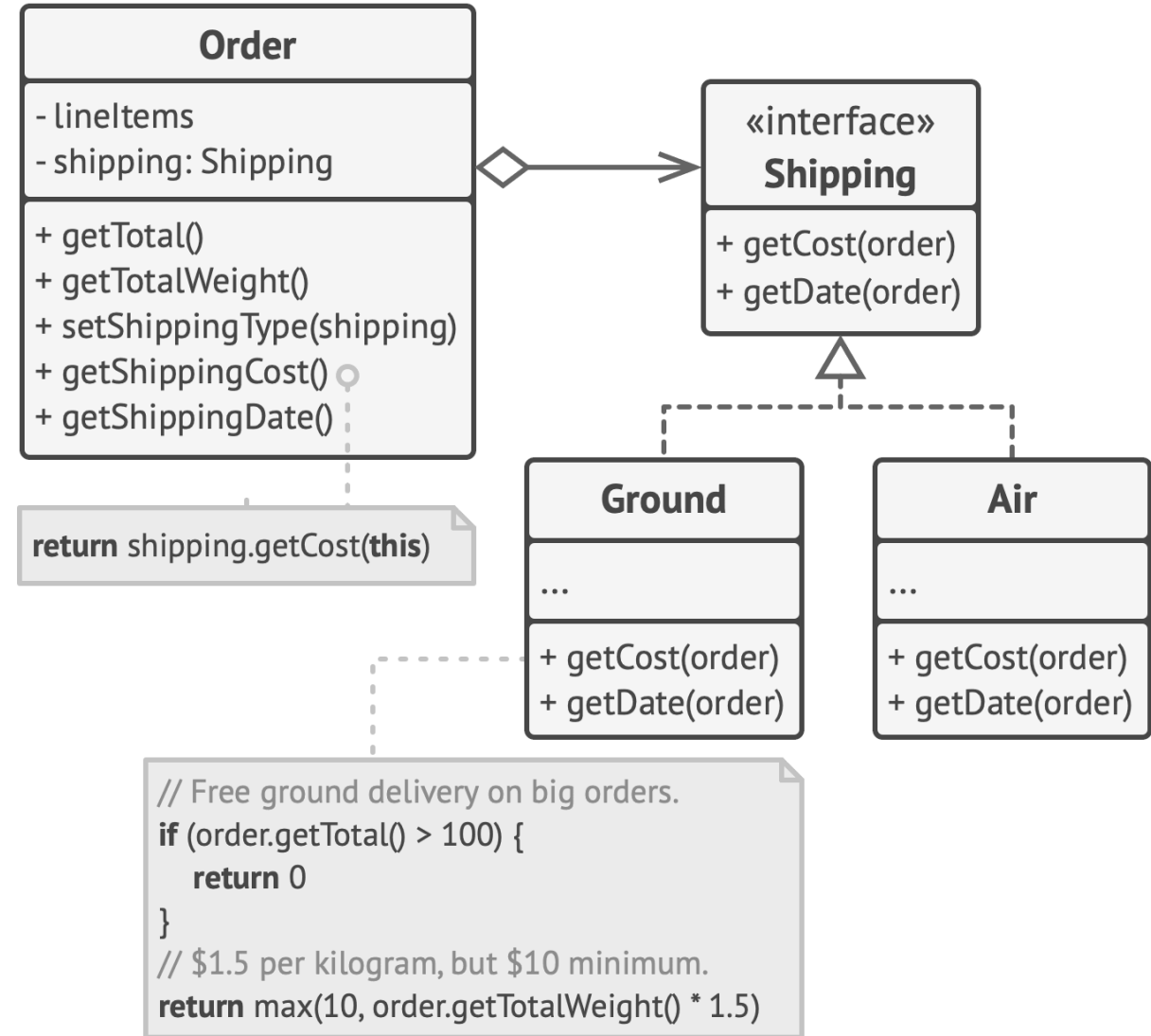
# SOLID 2: Open/Closed Principle (cont.)

- **Example**



```
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)
}
```



# SOLID 3: Liskov Substitution Principle

---

- When extending a class, ensure the capability of **passing 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
- Especially critical when developing libraries and frameworks
- **A set of checks**

## SOLID 3: Liskov Substitution Principle (cont.)

---

- (a) **Parameter types** in a method of a subclass should **match or be more abstract** than parameter types in the method of the superclass.
  - Example: **Base:** `feed(Cat c)` =>  
**Good:** `feed(Animal c)`, **Bad:** `feed(BengalCat c)`
- (b) The **return type** in a method of a subclass should **match or be a subtype** of the return type in the method of the superclass.
  - Example: **Base:** `adoptCat(): Cat` =>  
**Good:** `adoptCat(): BengalCat`, **Bad:** `adoptCat(): Animal`
- (c) Types of **exceptions** should **match or be subtypes** of the ones that the base method is already able to throw.
  - Built into most modern programming languages



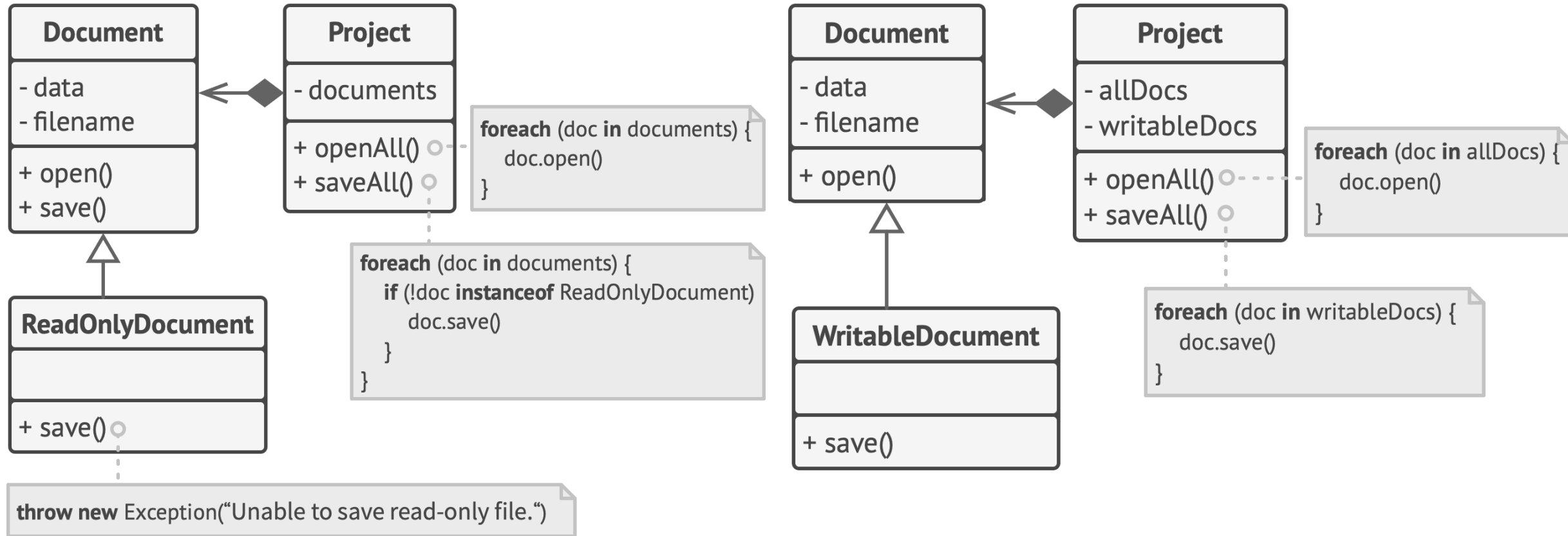
## **SOLID 3: Liskov Substitution Principle (cont.)**

---

- (d) A subclass **shouldn't strengthen pre-conditions**.
  - Example: a method with an int parameter
- (e) A subclass **shouldn't weaken post-conditions**.
  - Example: a method working with a database
- (f) **Invariants** of a superclass **must be preserved** (least formal rule of all)
  - Invariants: conditions in which an object makes sense
  - **Safest way** to extend a class: introduce new fields and methods
- (g) A subclass **shouldn't change values of private fields** of the superclass.

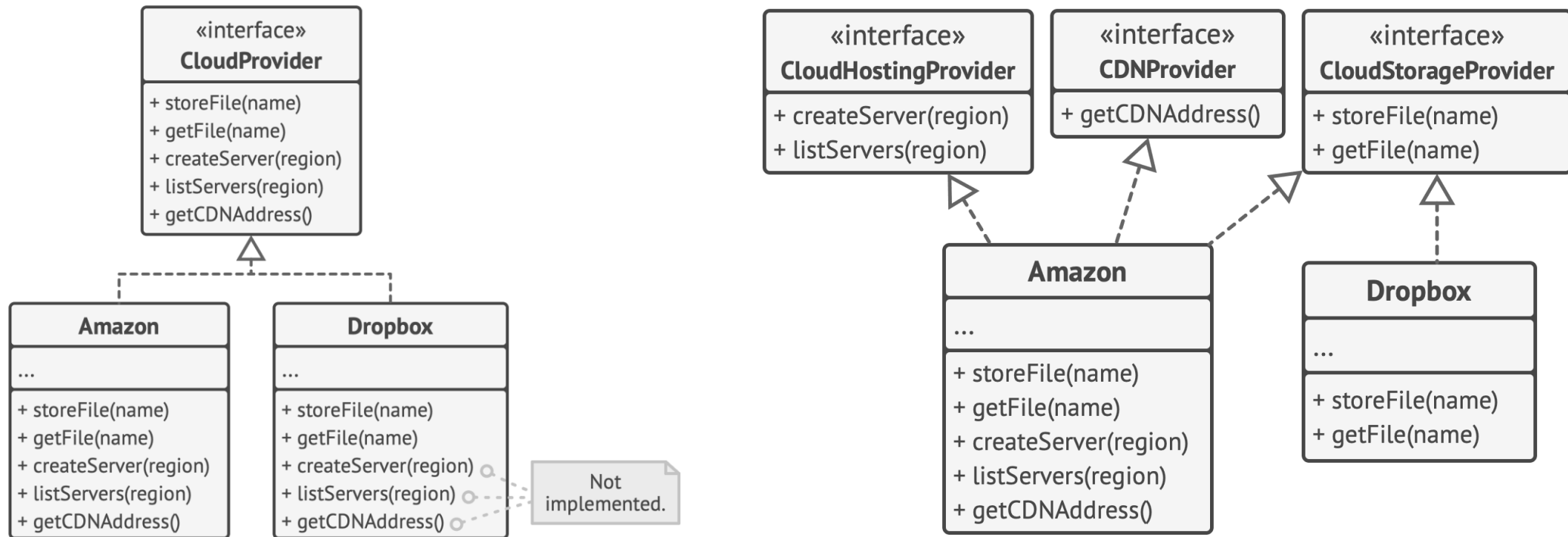
# SOLID 3: Liskov Substitution Principle (cont.)

## • Example



# SOLID 4: Interface Segregation Principle

- Clients **shouldn't be forced to depend on methods they do not use**
  - Make interfaces **narrow enough** that client classes don't have to implement behaviors they don't need
  - **Break down** “fat” interfaces into more granular and specific ones



# SOLID 5: Dependency Inversion Principle

---

- **High-level classes shouldn't depend on low-level classes, and both should depend on abstractions**
- Low-level classes vs. high-level classes
- Problem: business logic classes tend to become dependent on primitive low-level classes
- **Suggestion: changing the direction of dependency**
- **Approach**
  1. Describe interfaces for low-level operations that high-level classes rely on, preferably in business terms
  2. Make high-level classes dependent on the interfaces, resulting in a softer dependency
  3. Low-level classes implement the interfaces, dependent on the business logic level

# SOLID 5: Dependency Inversion Principle (cont.)

- **Example**

