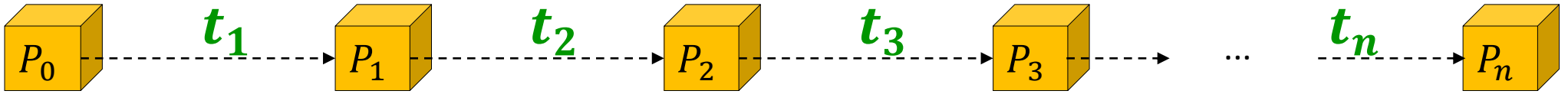
(6) Design Patterns #1

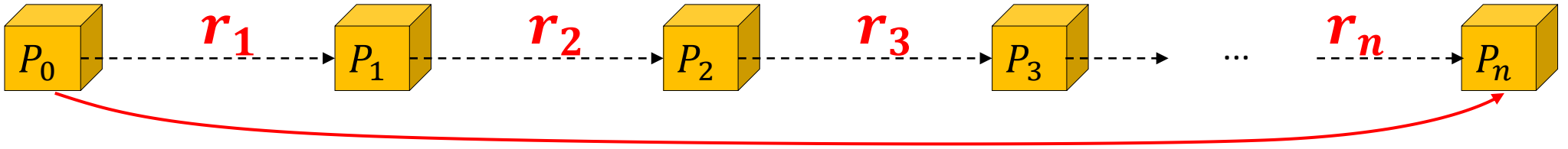
A View of Program Evolution

- A program is developed through a series of transformations



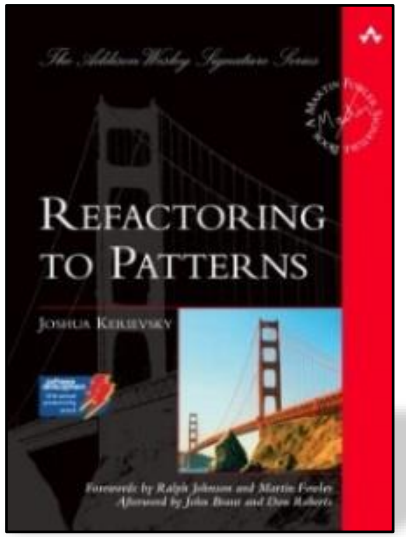
An Automated View of Program Evolution

- A program is developed through a series of refactorings



Design Pattern = ∑ refactorings

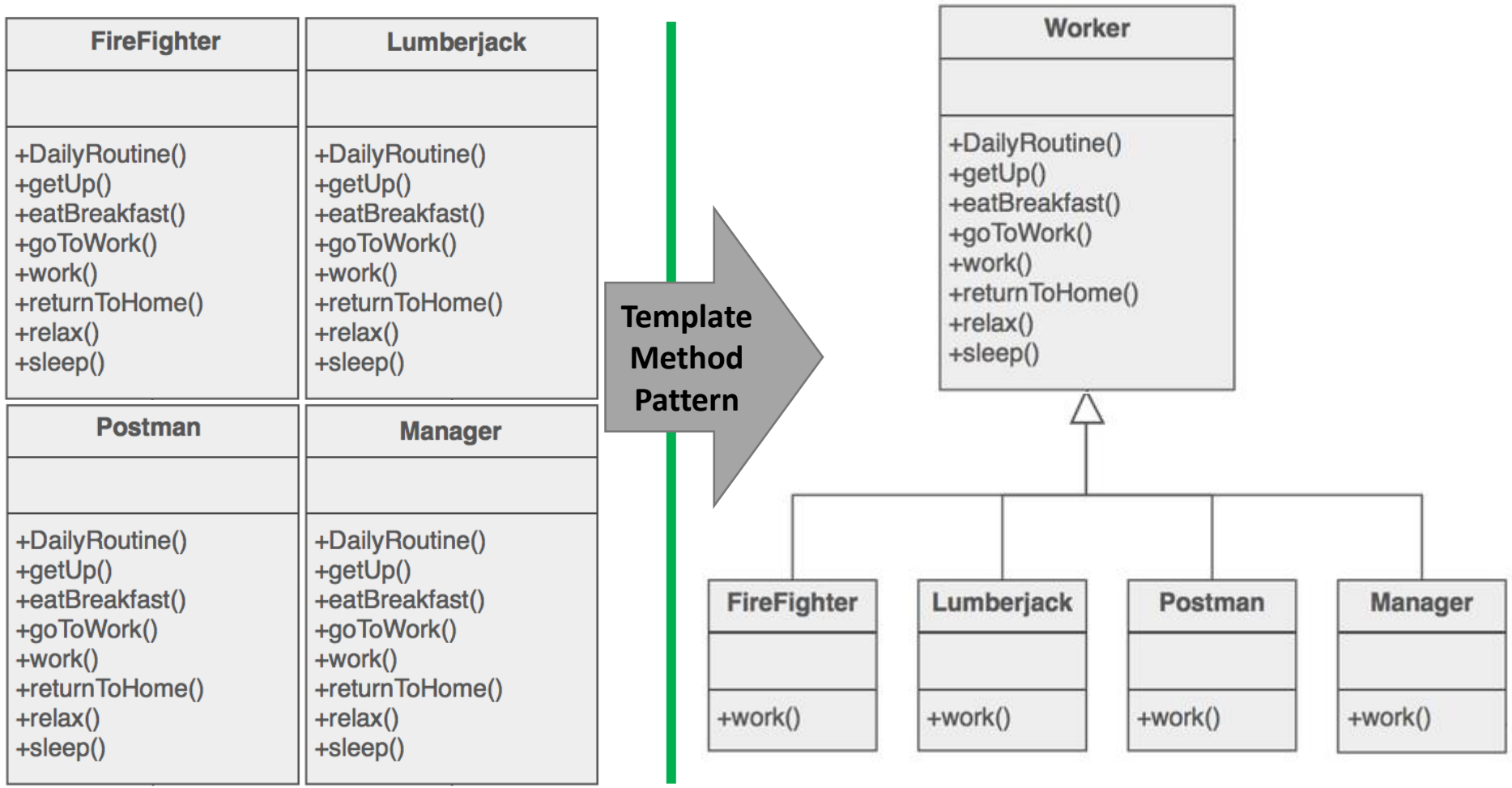
- These ideas were introduced over 15 years ago



Design Patterns

- Design Patterns define reusable solutions to design problems in object-oriented programming
  - Relationships and interactions between classes or objects

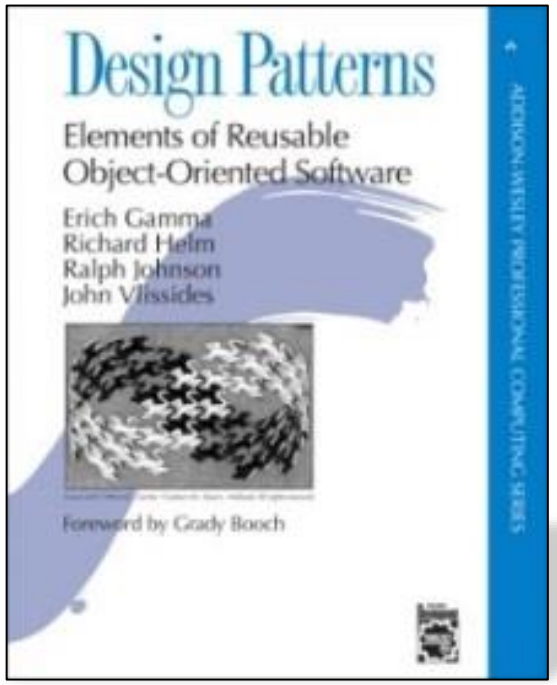
Template Method Pattern



GoF's Design Patterns (1994)

- 23 patterns that expert OO programmers use
  - e.g., template method, singleton, visitor, factory, etc.

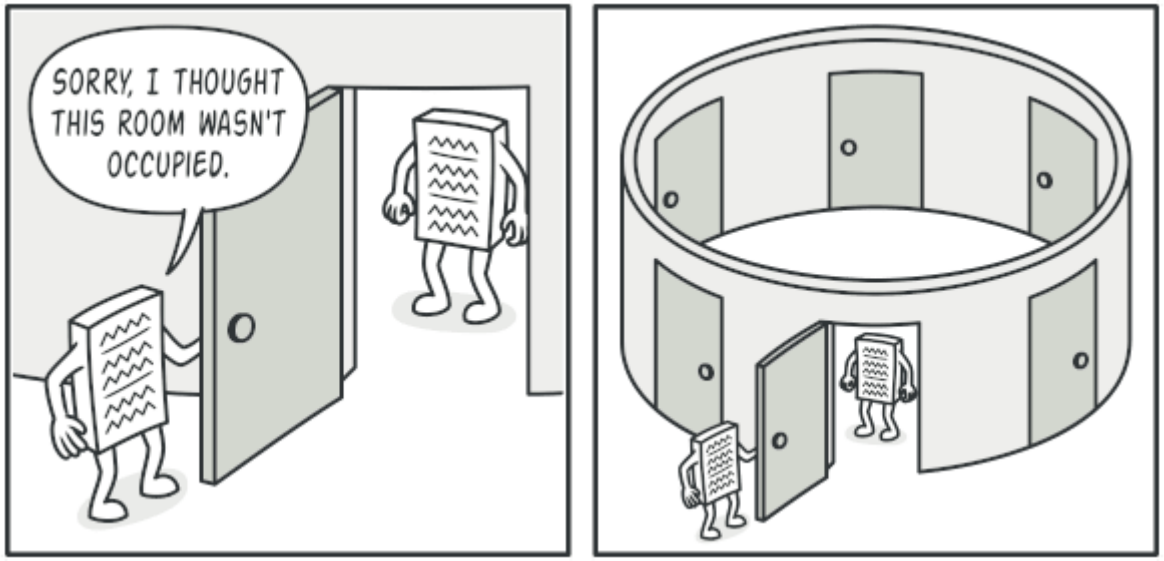
- Goal: non-expert programmers can design "like experts"



Singleton

Singleton

- Singleton pattern ensures that a class has only one instance



Clients may not realize that they are working with the same object all the time

Example

```
public class Earth {
    private Earth() {}

    private static Earth instance = null;

    public static Earth getInstance() {
        if (instance == null)
            instance = new Earth();
        return instance;
    }
}
```

Exercise

- A program that shows Singleton class A has only one instance

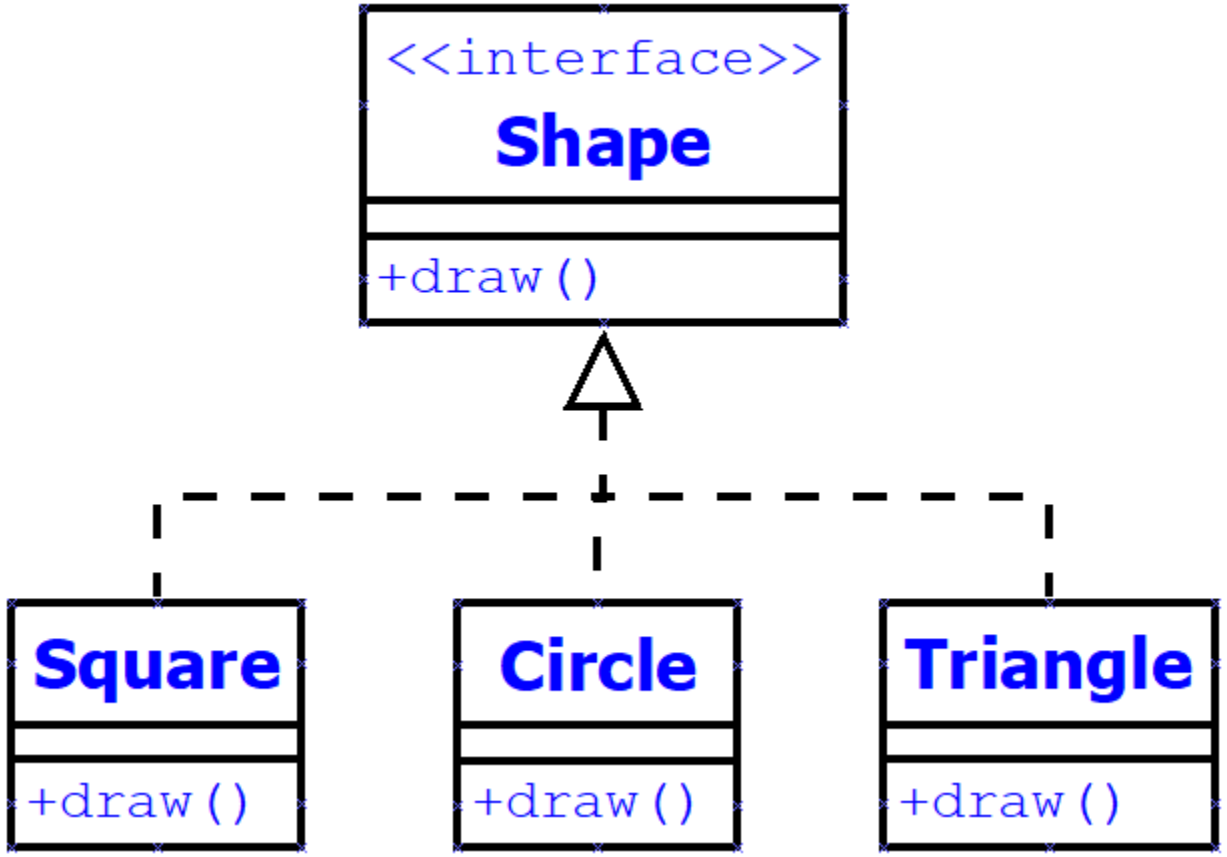
```
public class A {
    public String s;

    private A() {}

    public static A getInstance() {
        if (instance == null)
            instance = new A();
        return instance;
    }

    private static A instance = null;
}
```

```
A x = A.getInstance();
x.s = "Hi";
A y = A.getInstance();
y.s = "Hello";
A z = A.getInstance();
z.s = "Bye";

System.out.println("String from x is " + x.s);
System.out.println("String from y is " + y.s);
System.out.println("String from z is " + z.s);

y.s = "Interesting!";

System.out.println("String from x is " + x.s);
System.out.println("String from y is " + y.s);
System.out.println("String from z is " + z.s);
```

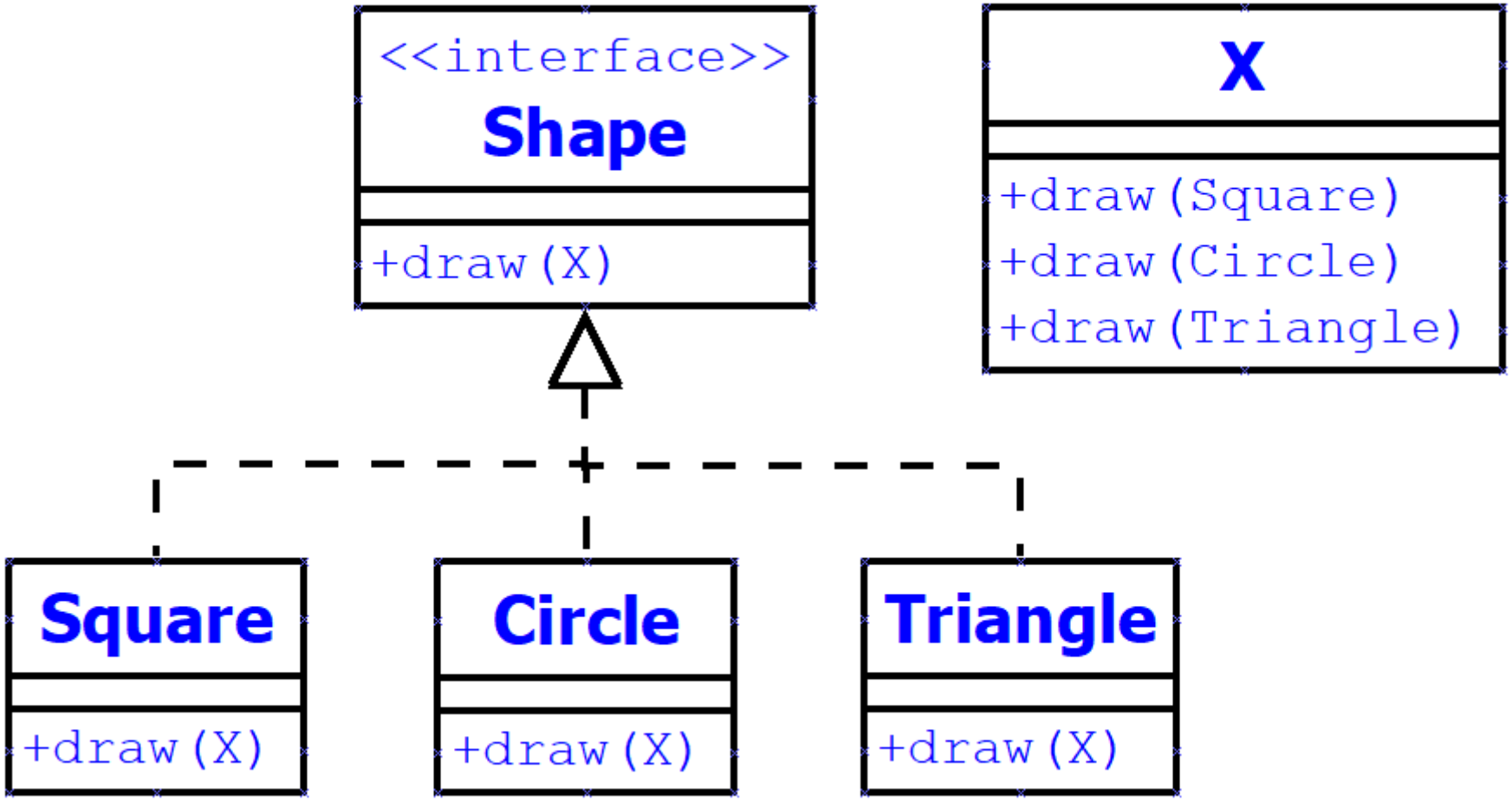
Visitor

Motivation

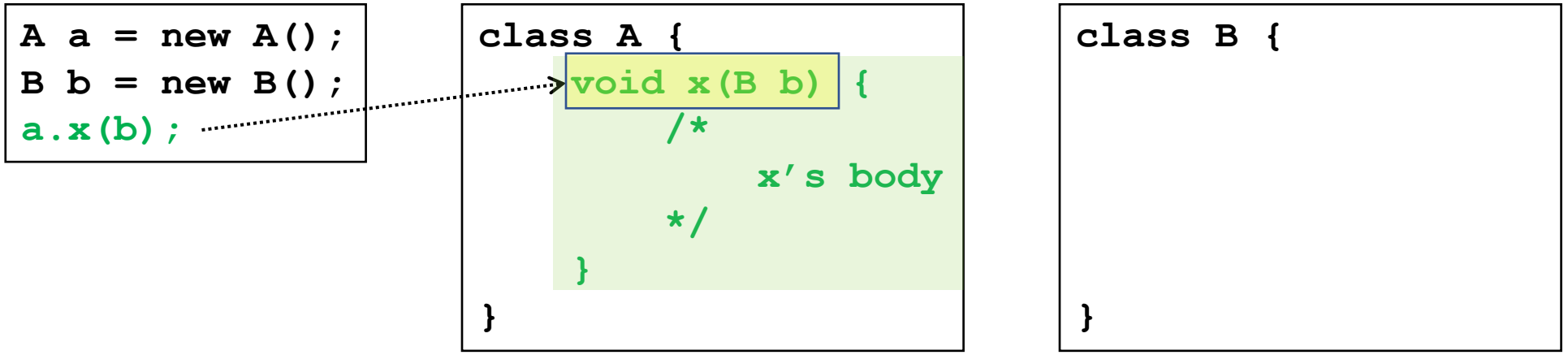
- Can you separate the graphic application's structure from operation code?



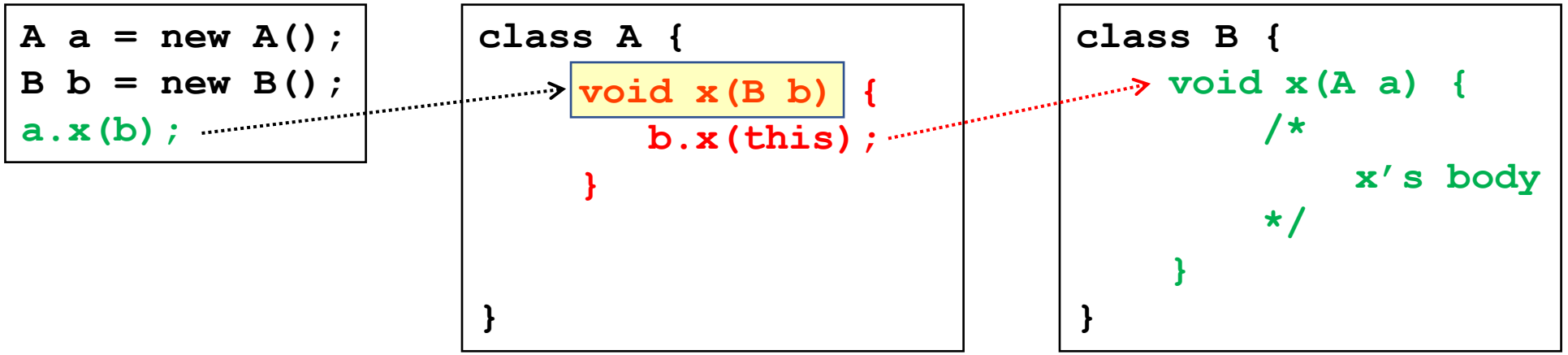
Separation of Structure and Implementation



Recap: Delegate



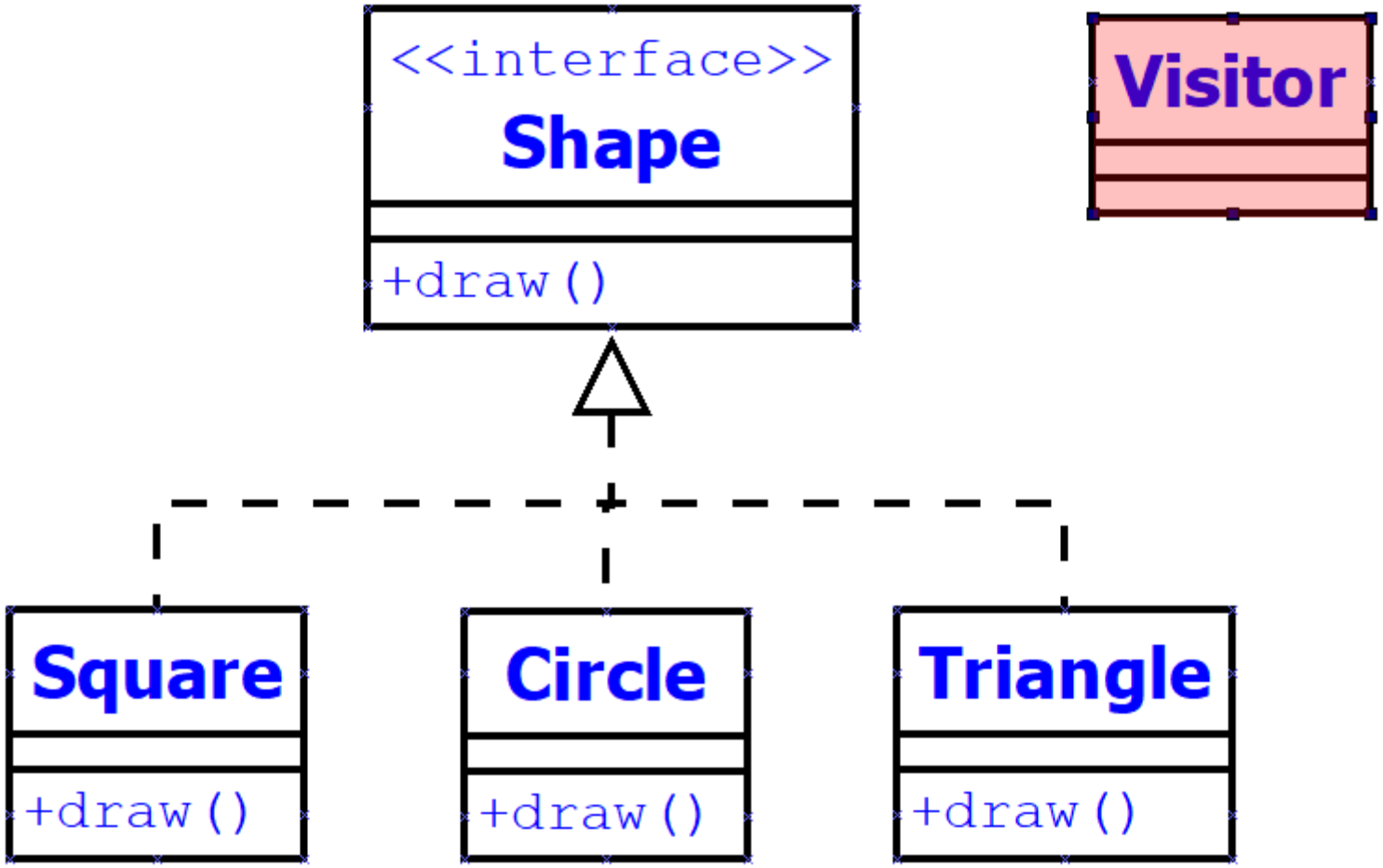
move and leave a delegate behind ⇓



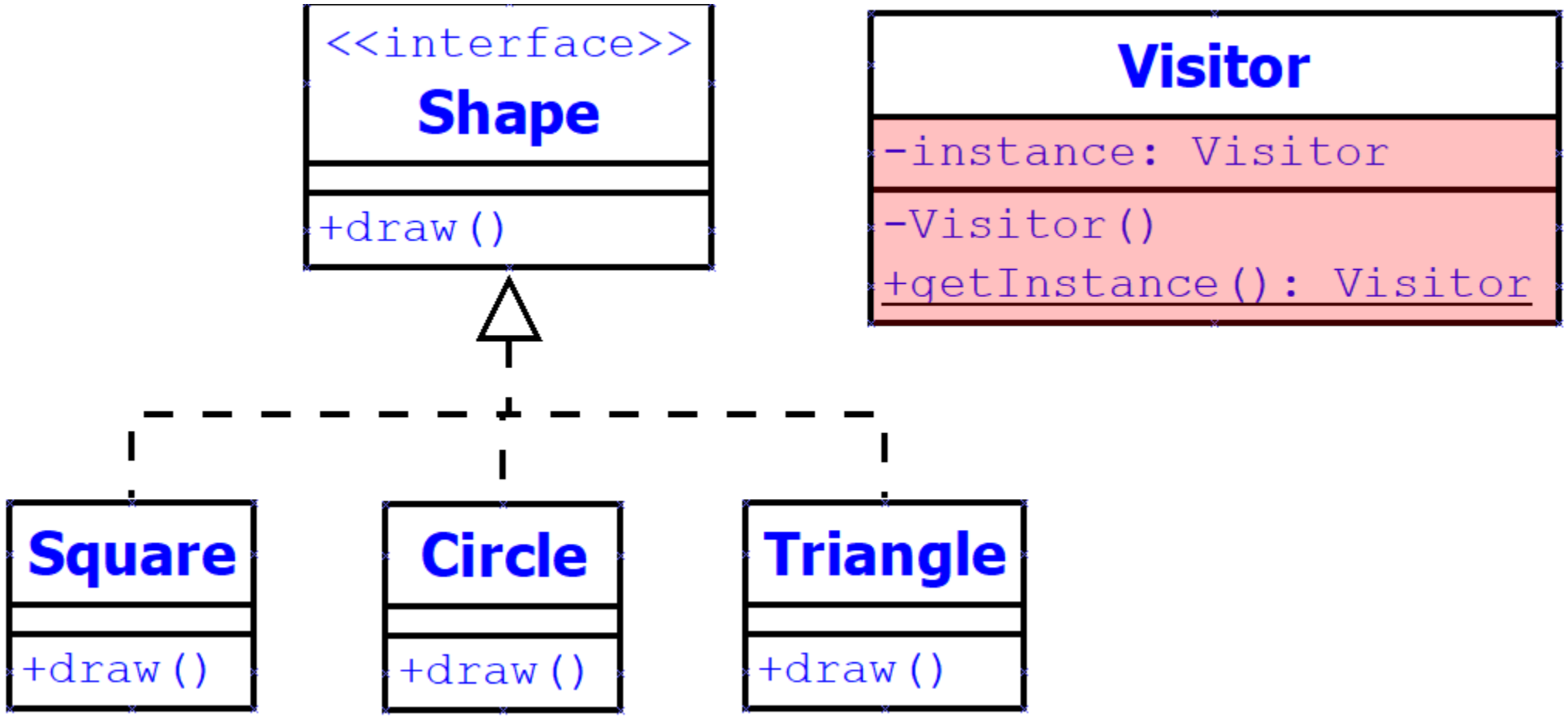
Basic Steps

Shape.zip

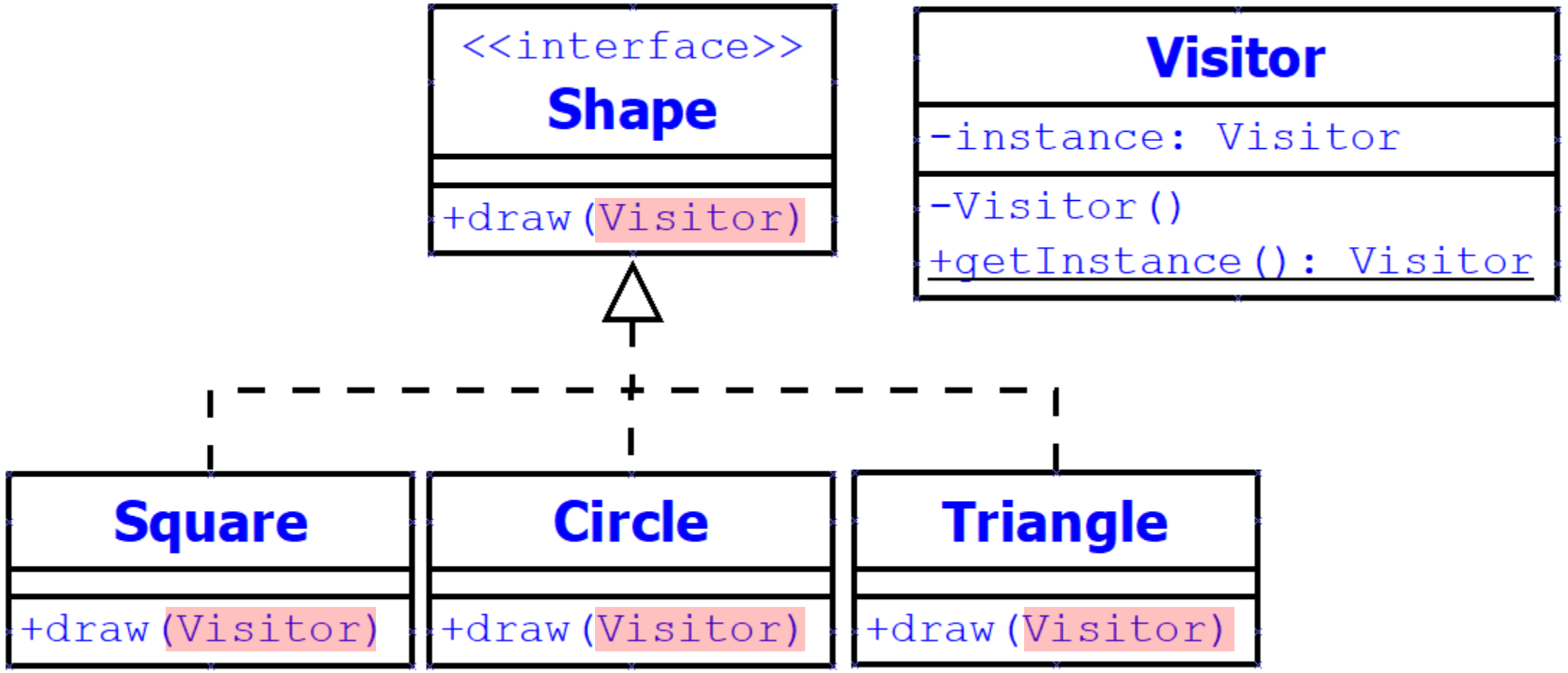
1. Create a Visitor Class



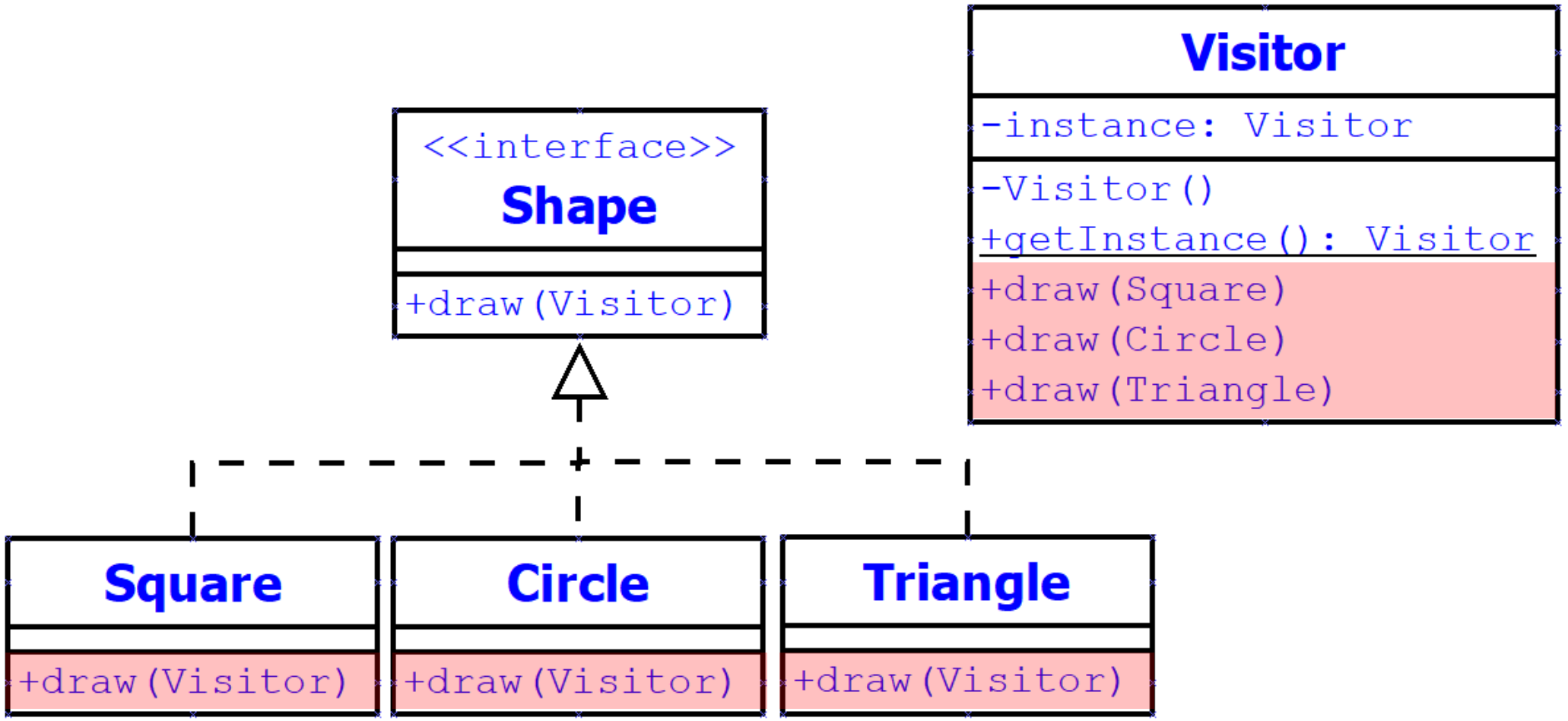
2. Make Visitor “Singleton”



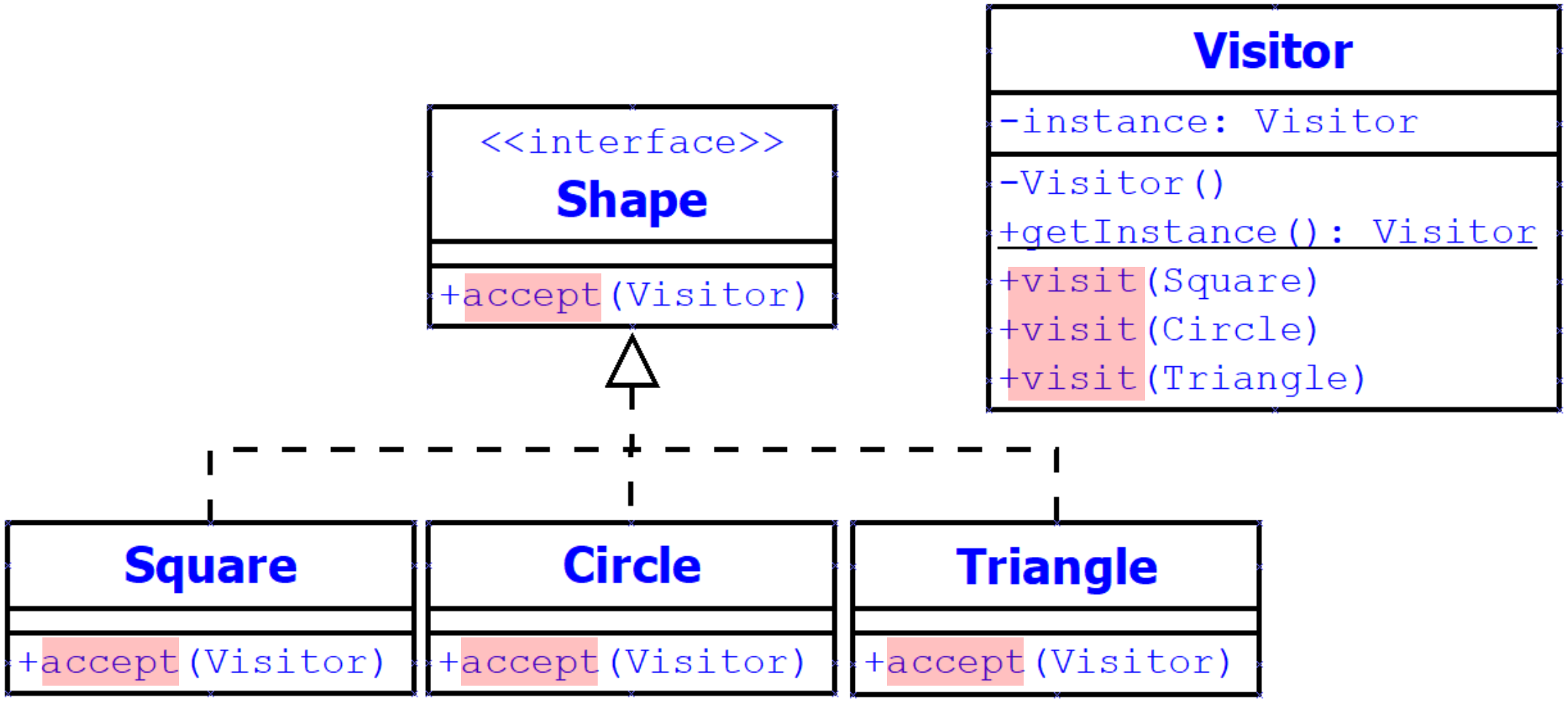
3. Add a Visitor-Type Parameter



4. MiMr via Parameter with Leaving a Delegate



5. Rename Methods to Visit and Accept

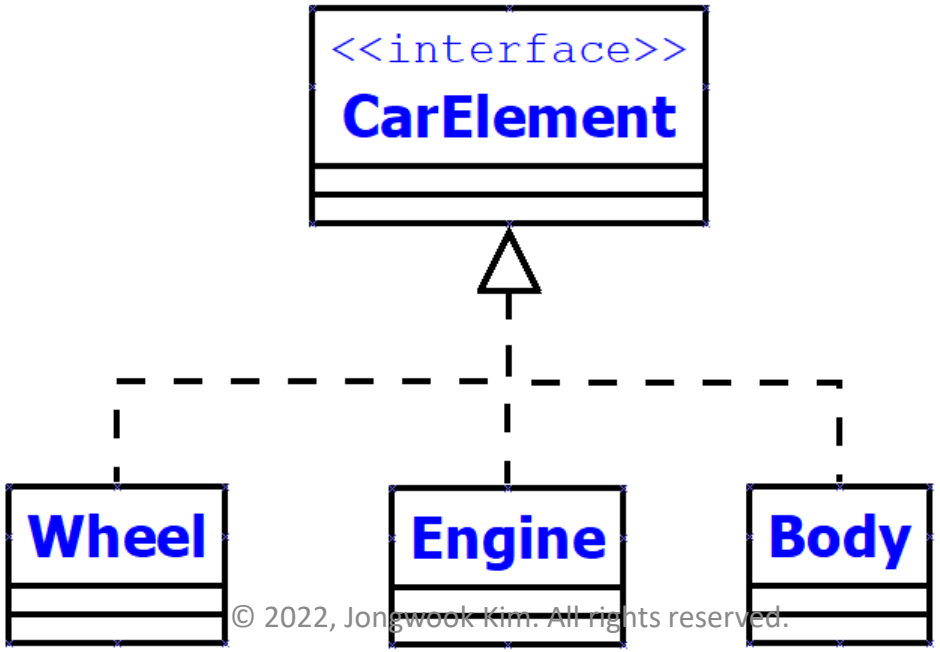


Run-time Polymorphic Visitors

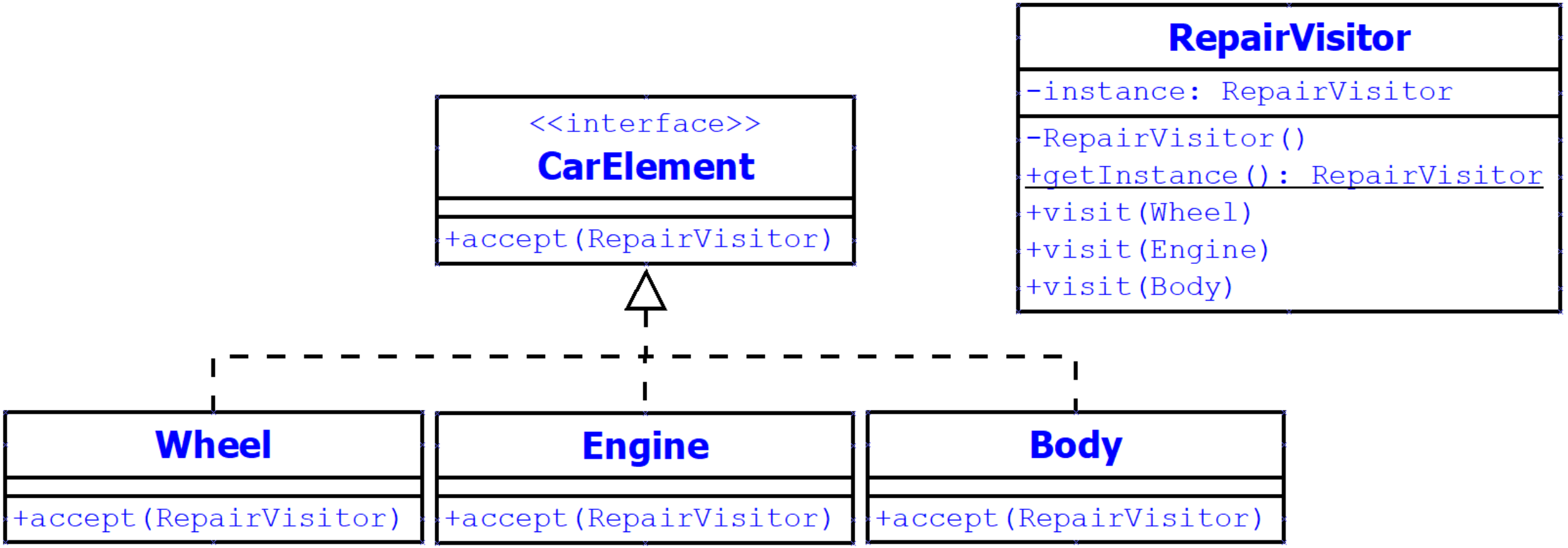
CarElement.zip

Example: CarElement

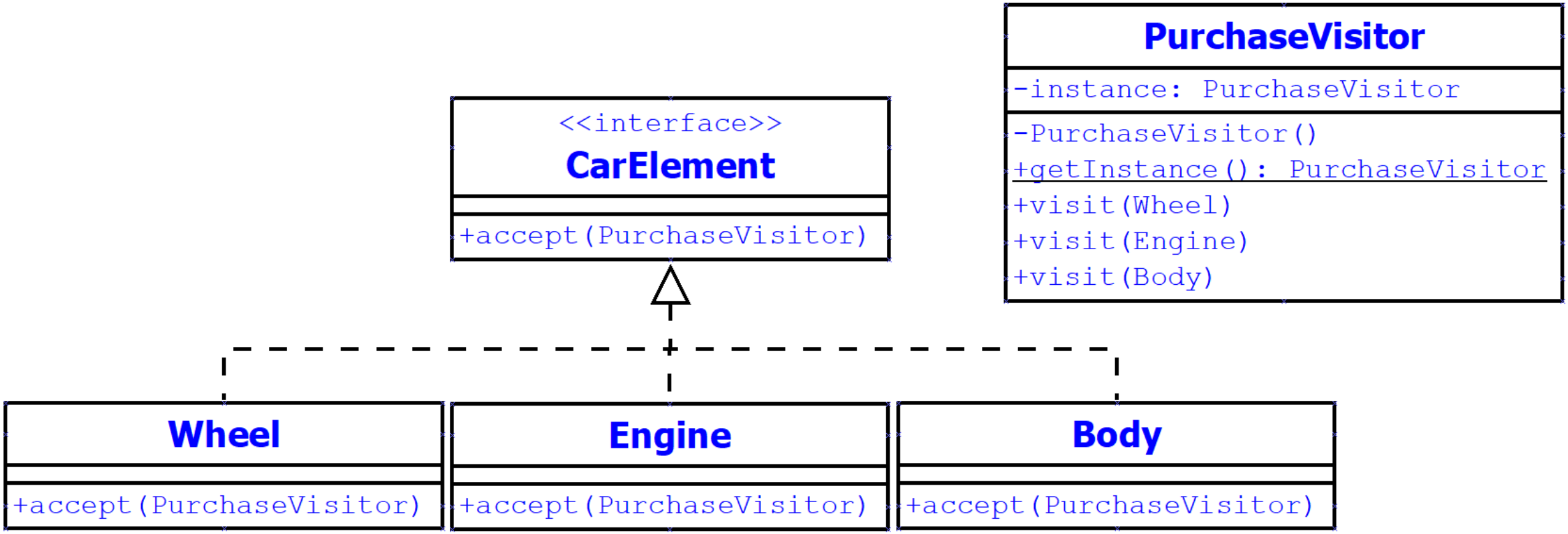
- Suppose that we have interface CarElement, which is implemented by classes Wheel, Engine, Body
  - Can you separate the car element structure from operation code?
  - We need “repair” and “purchase” operations on the car elements



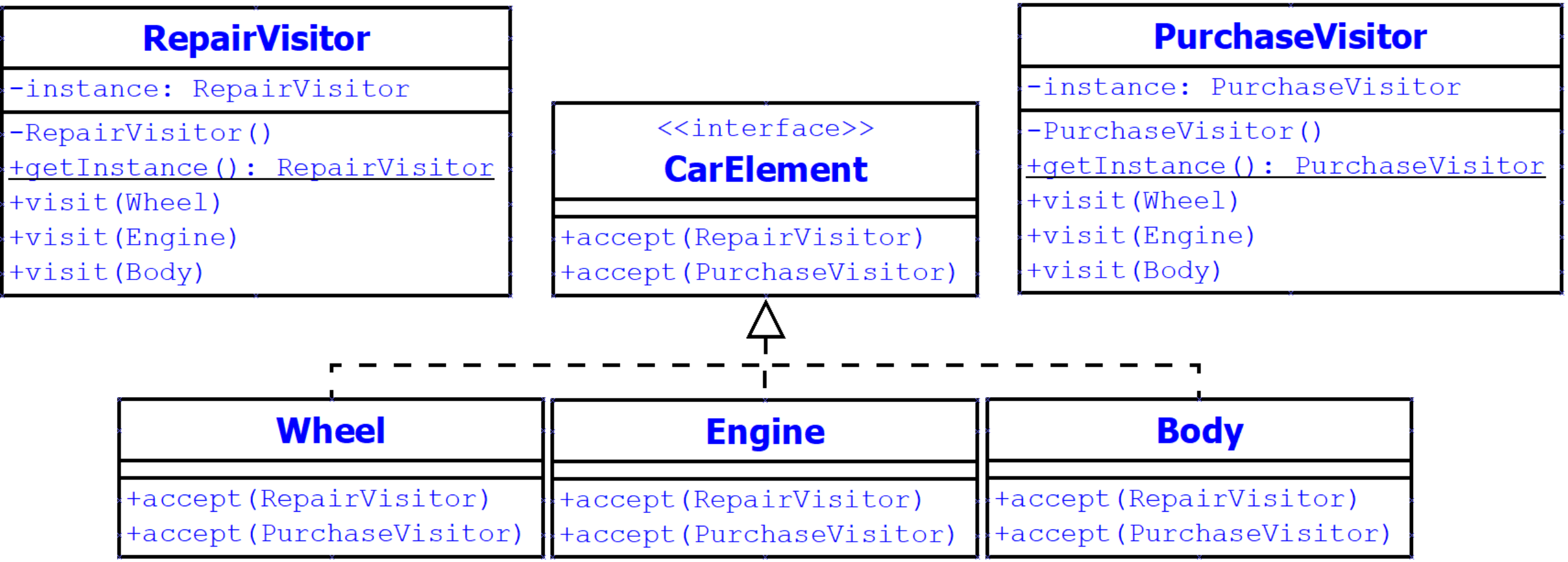
RepairVisitor



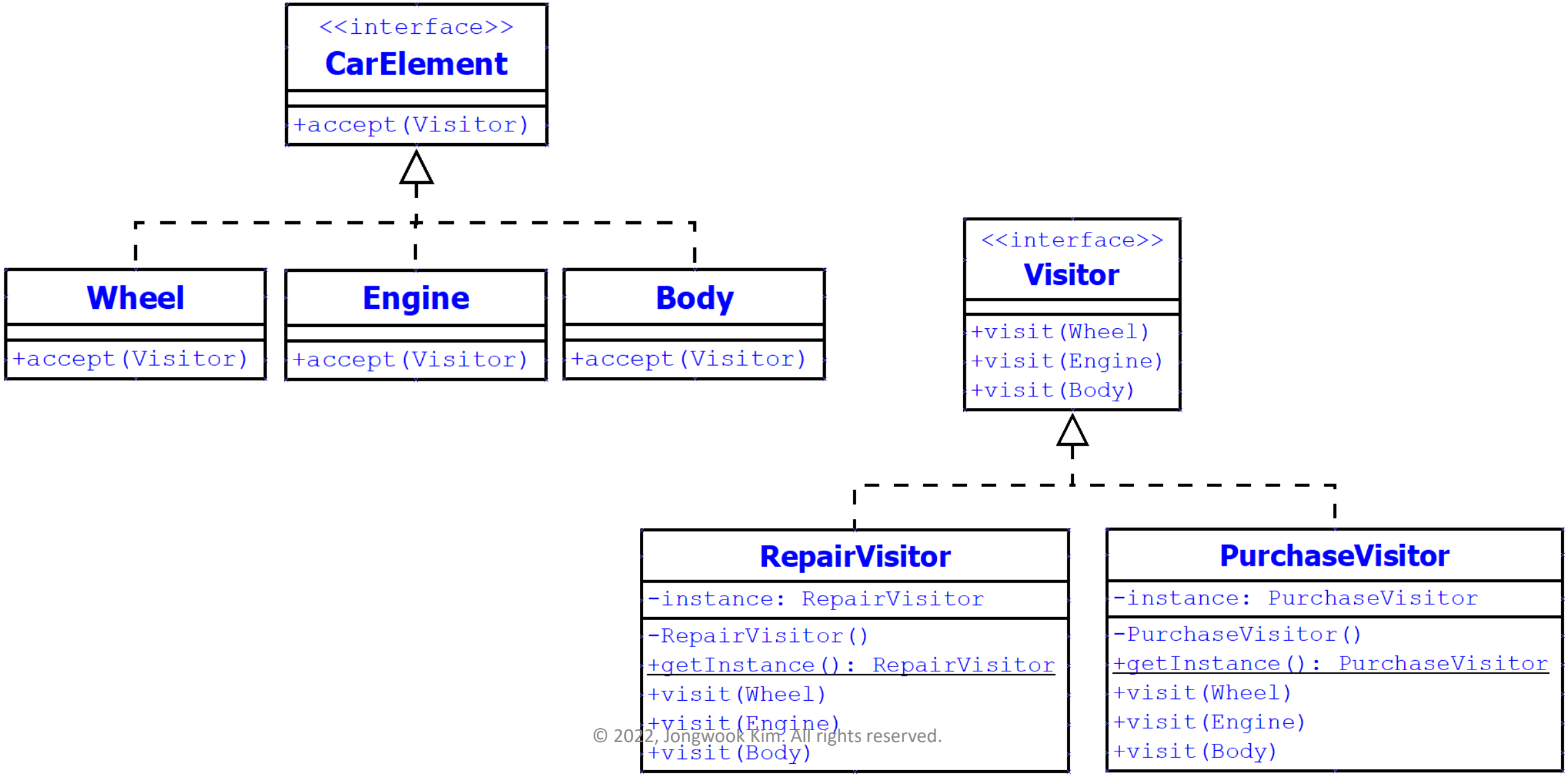
PurchaseVisitor



Code Redundancy!



Run-time Polymorphic Visitors



Advantages of Visitor Pattern

- It allows to add operations to a structure without changing the structure itself
- The code for operations is centralized