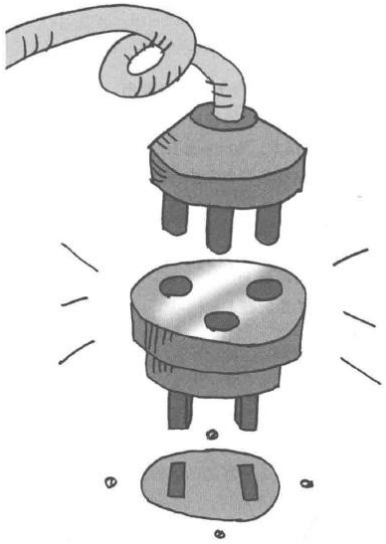
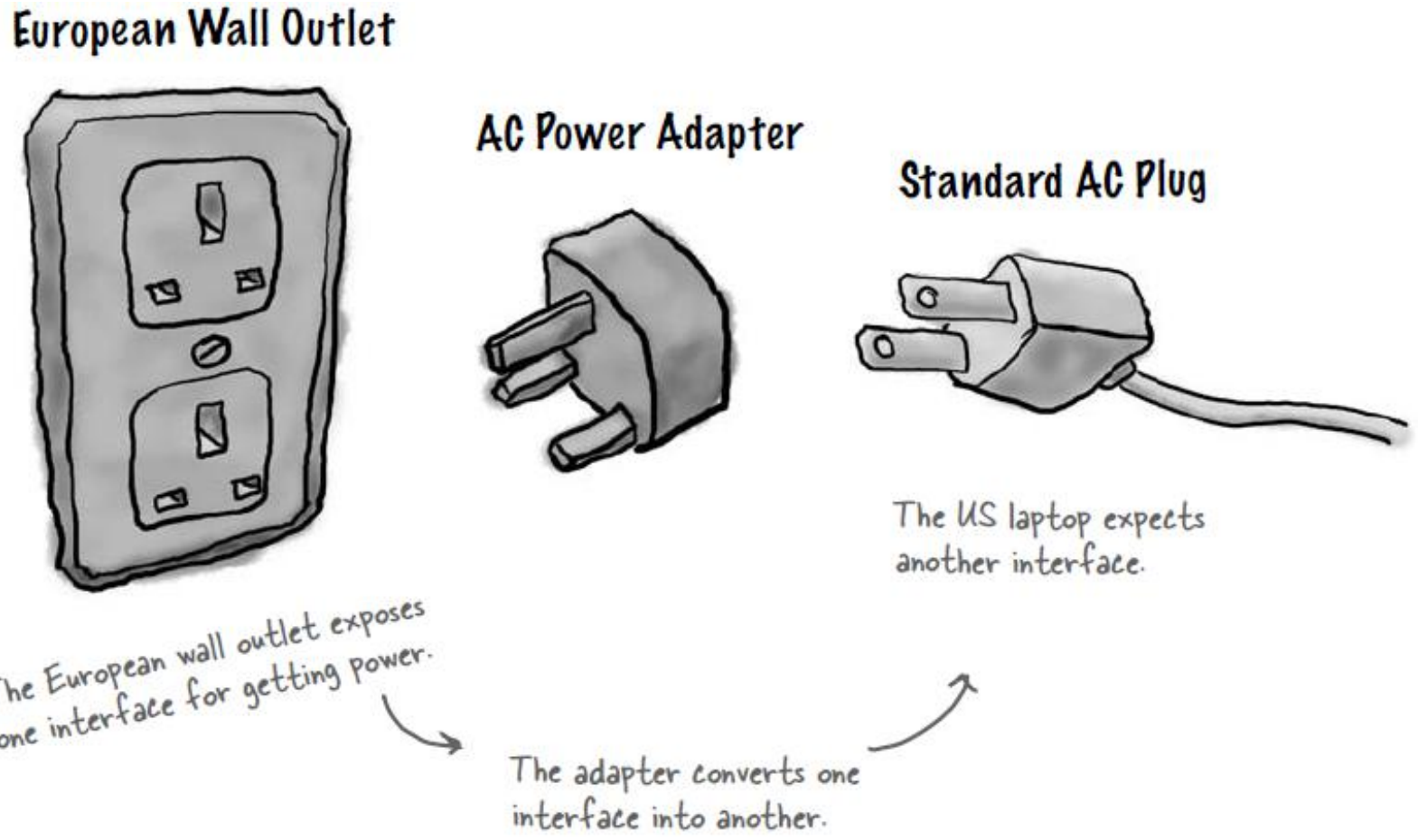
Adapter Pattern (适配器, Structural Pattern)



Kai SHI

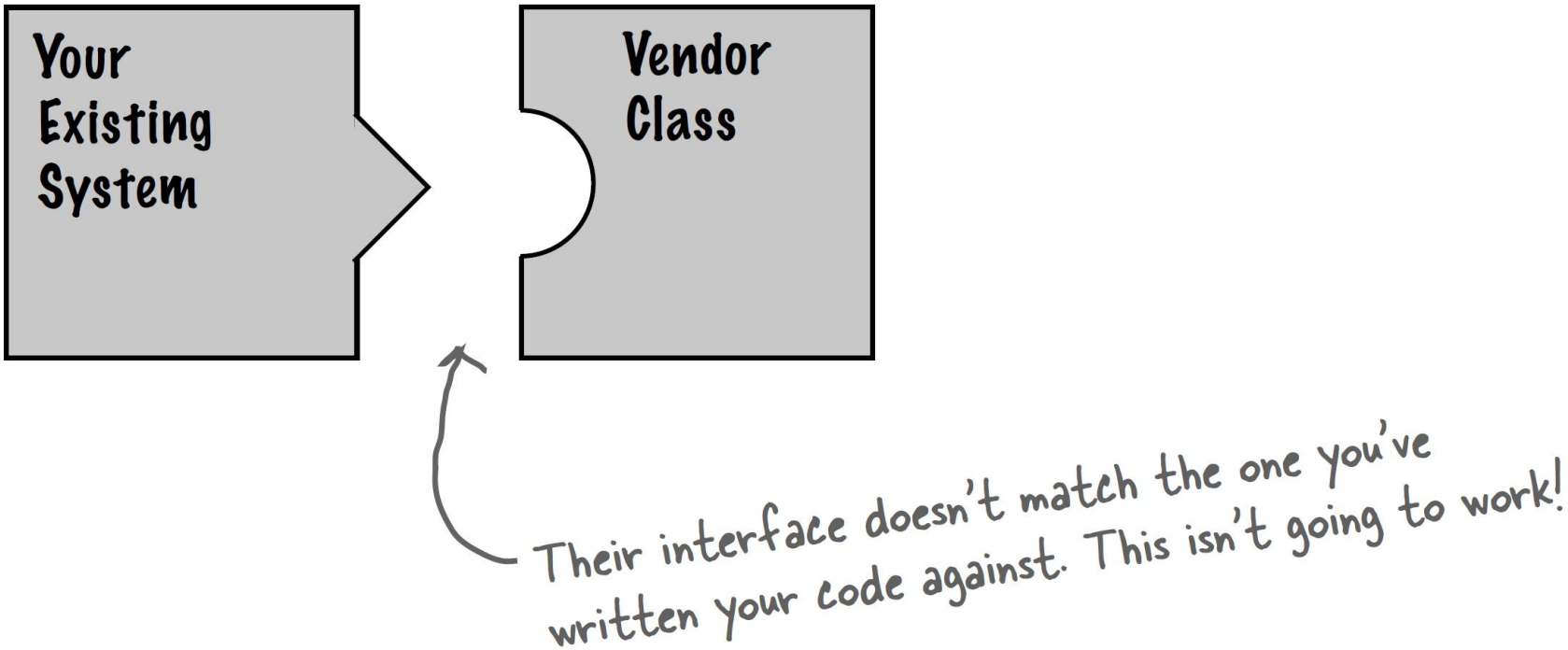
Adapters

The **adapter changes the interface** of the outlet into one that your laptop expects



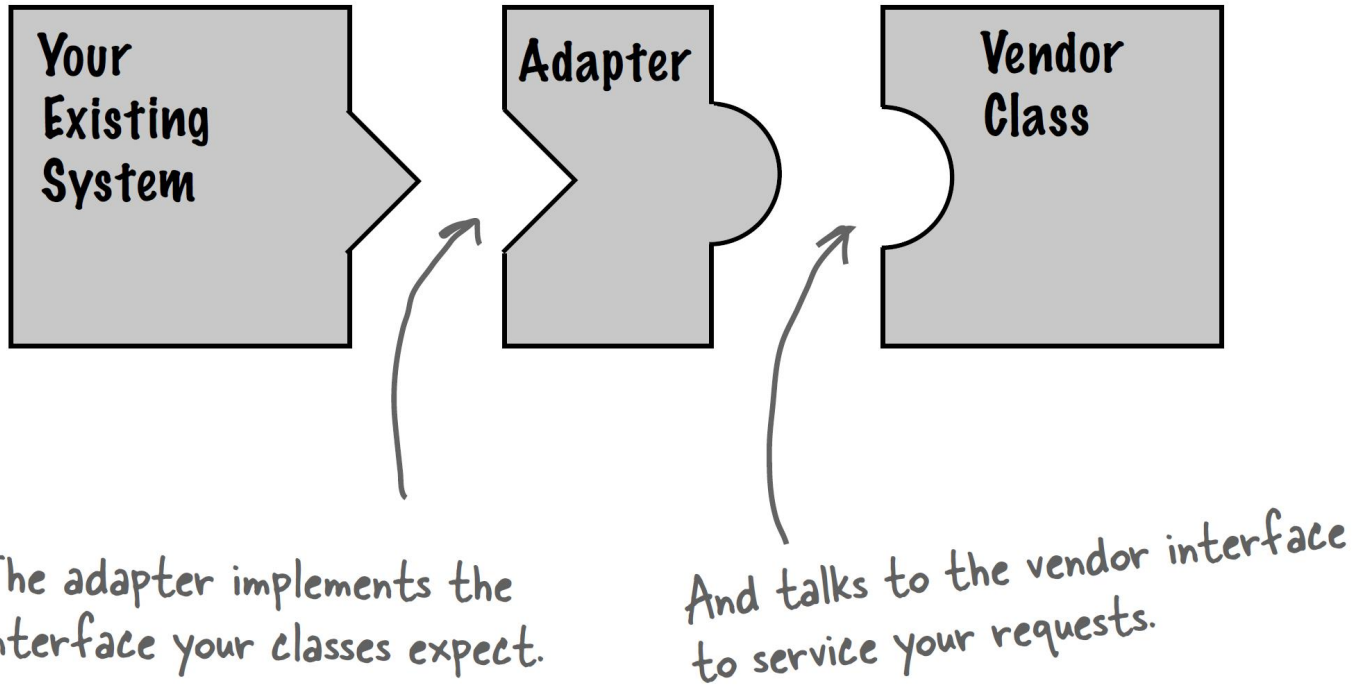
Object Oriented Adapters (1/3)

- You've got an existing software system that you need to work a new vendor class library into, but the new vendor designed their interfaces differently than the last vendor.
- You don't want to solve the problem by changing your existing code and you can't change the vendor's code.



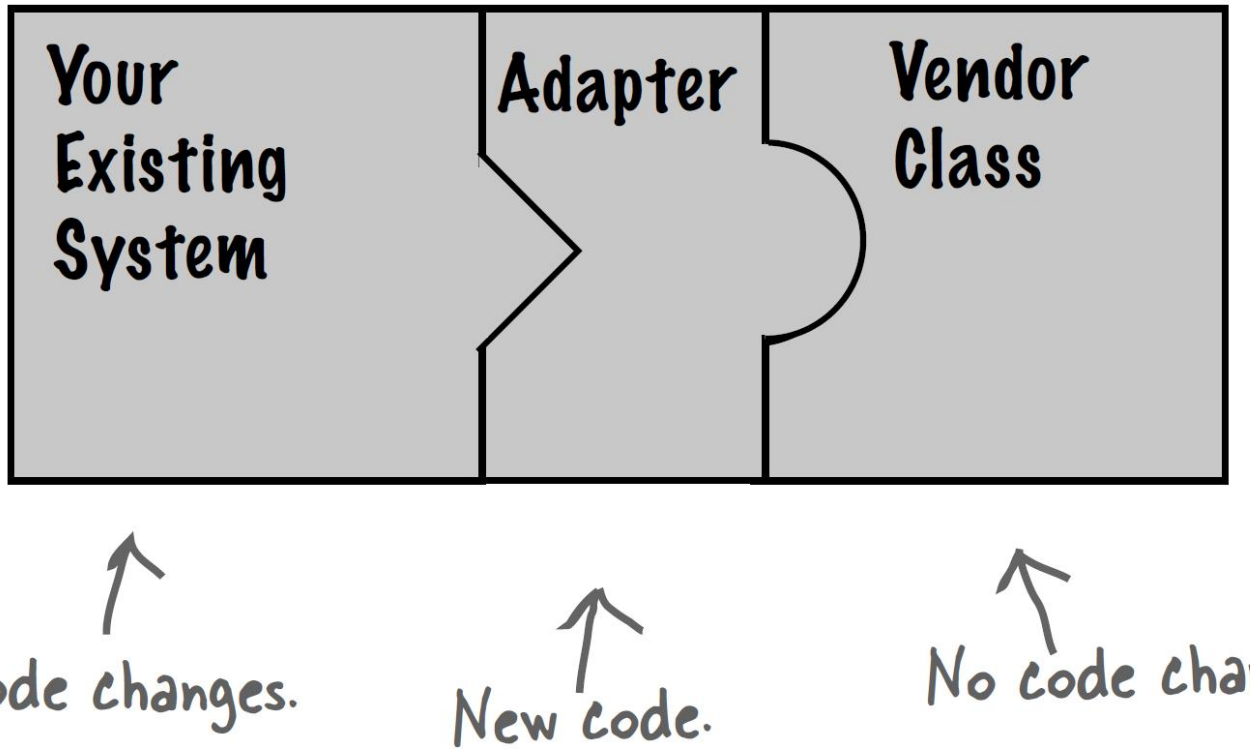
Object Oriented Adapters (2/3)

- You can write a class that adapts the new vendor interface into the one you're expecting.



Object Oriented Adapters (3/3)

- The adapter acts as the middleman by receiving requests from the client and converting them into requests that make sense on the vendor classes.



Example: If it walks like a duck and quacks like a duck, then it ~~must~~ might be a ~~duck~~ turkey wrapped with a duck adapter...

```
public interface Duck {
    public void quack();
    public void fly();
}
```

This time around, our ducks implement a Duck interface that allows Ducks to quack and fly.

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck... and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

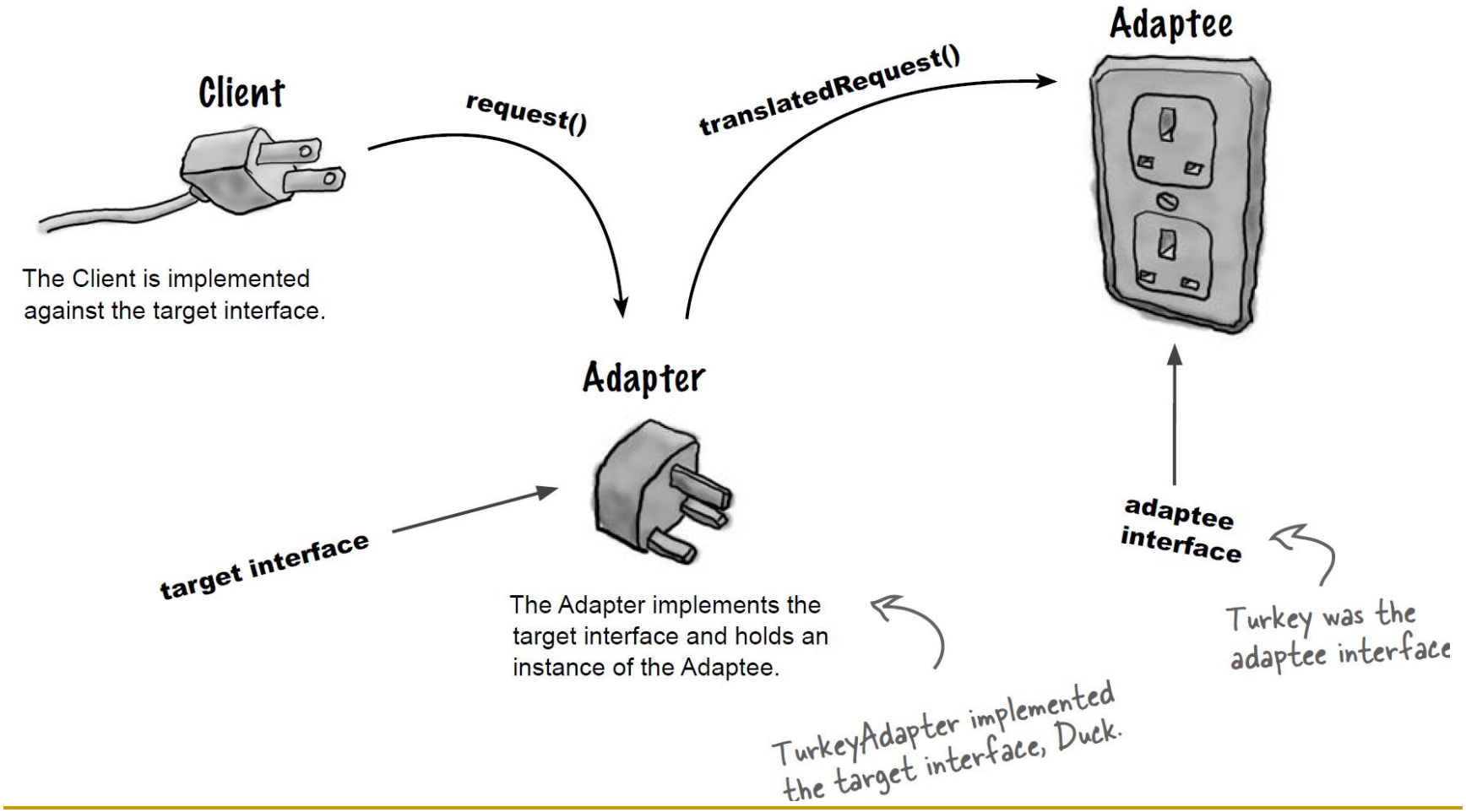
Then, let's test the Turkey: make it gobble, make it fly.

Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

The Adapter Pattern Explained



How the Client uses the Adapter?

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
3. The client receives the results of the call and never knows there is an adapter doing the translation.

Adapter Pattern

- Intent
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As
  - Wrapper (包装类)

Can you remember me? Decorator is also a kind of wrapper.

Motivation

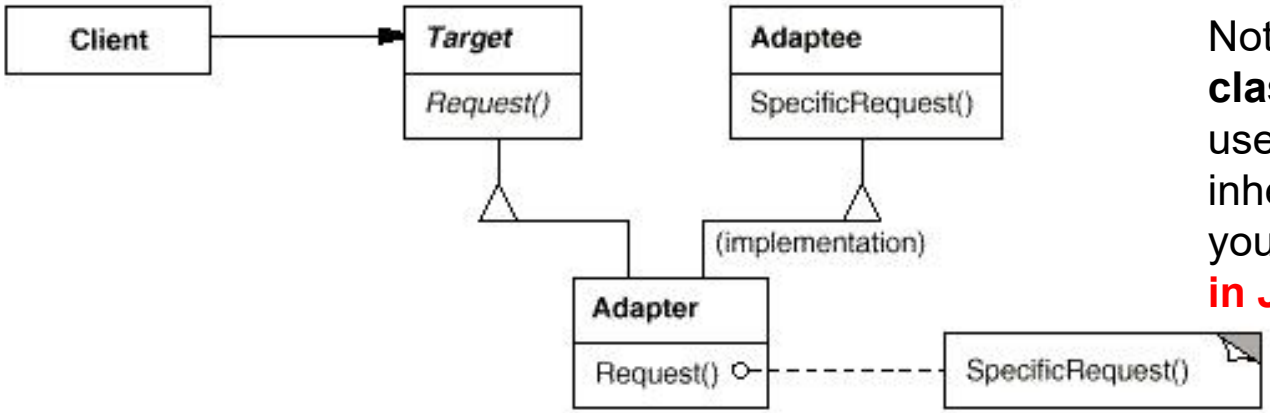
- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Applicability: Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

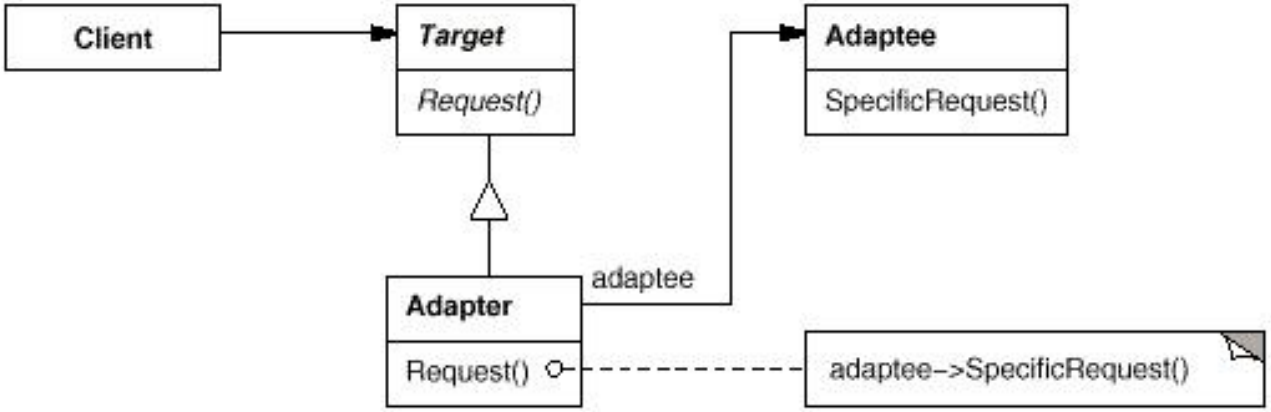
Structure

- A **class adapter** (类适配器) uses multiple inheritance to adapt one interface to another:



Note: the **class adapter** uses multiple inheritance, so you **can't do it in Java**...

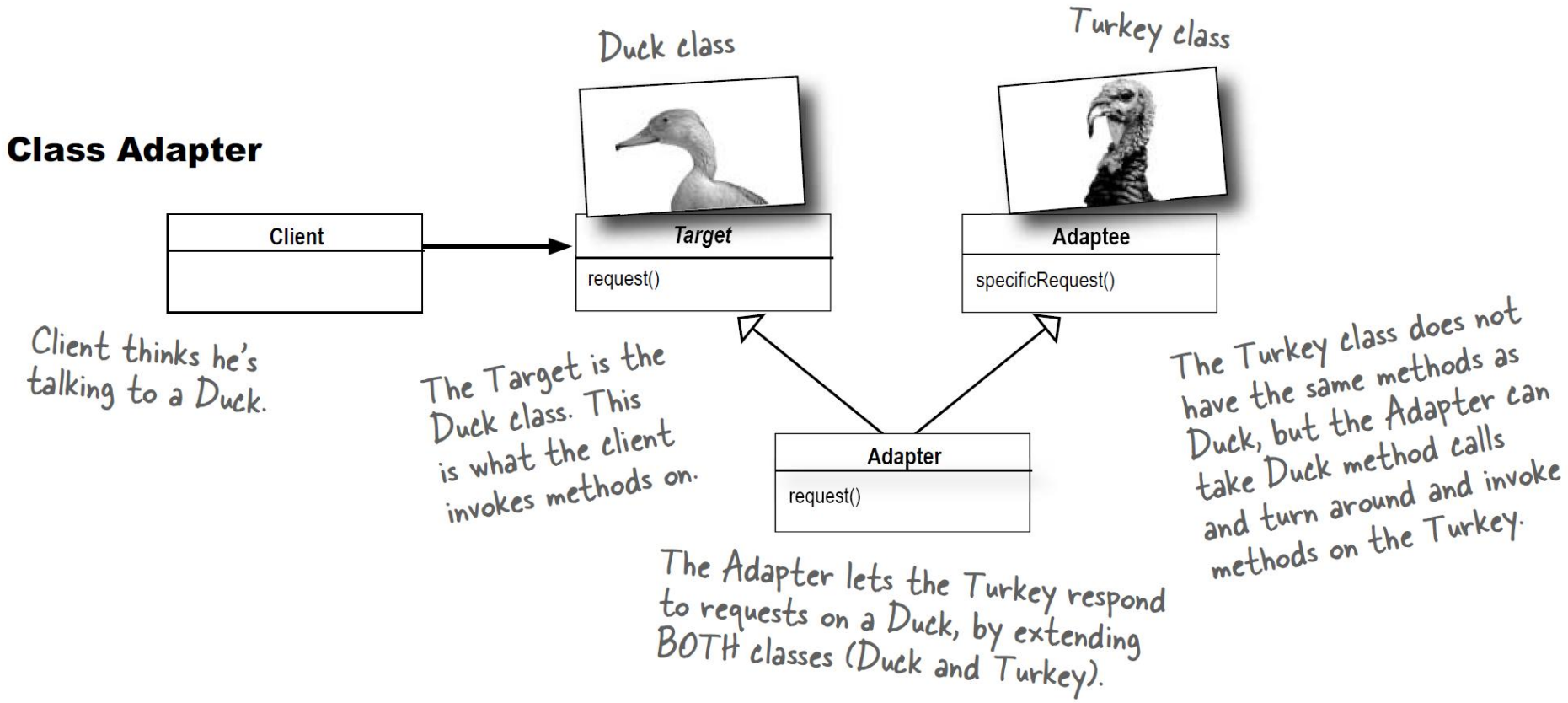
- An **object adapter** (对象适配器) relies on object composition:



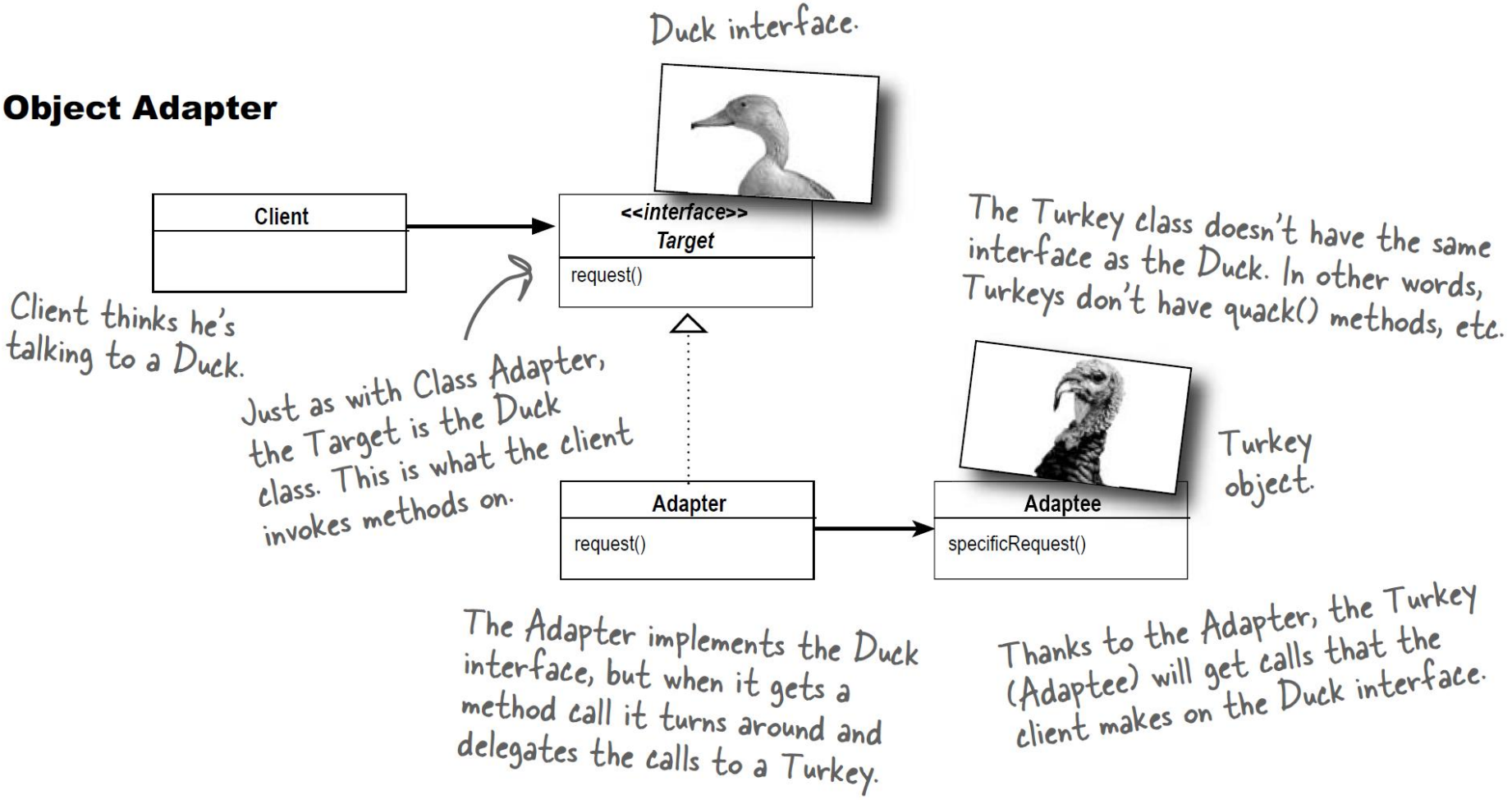
Participants

- Target: Defines the domain-specific interface that Client uses. It should be an interface;
- Client: Collaborates with objects conforming to the Target interface;
- Adaptee: Defines an existing interface that needs adapting, could be an interface, or abstract class, or class;
- Adapter: Adapts the interface of Adaptee to the Target interface.

Note: the **class adapter** uses multiple inheritance, so you **can't do it in Java**...



Object Adapter



Consequences: Class Adapter

- Adapting Adaptee to Target by committing (托付) to a concrete Adapter class;
  - Adapter could override some of Adaptee's behavior;
- A class adapter won't work when we want to adapt a class and all its subclasses;
- Introducing only one concrete Adapter class, there is only one way making client access the Adaptee.

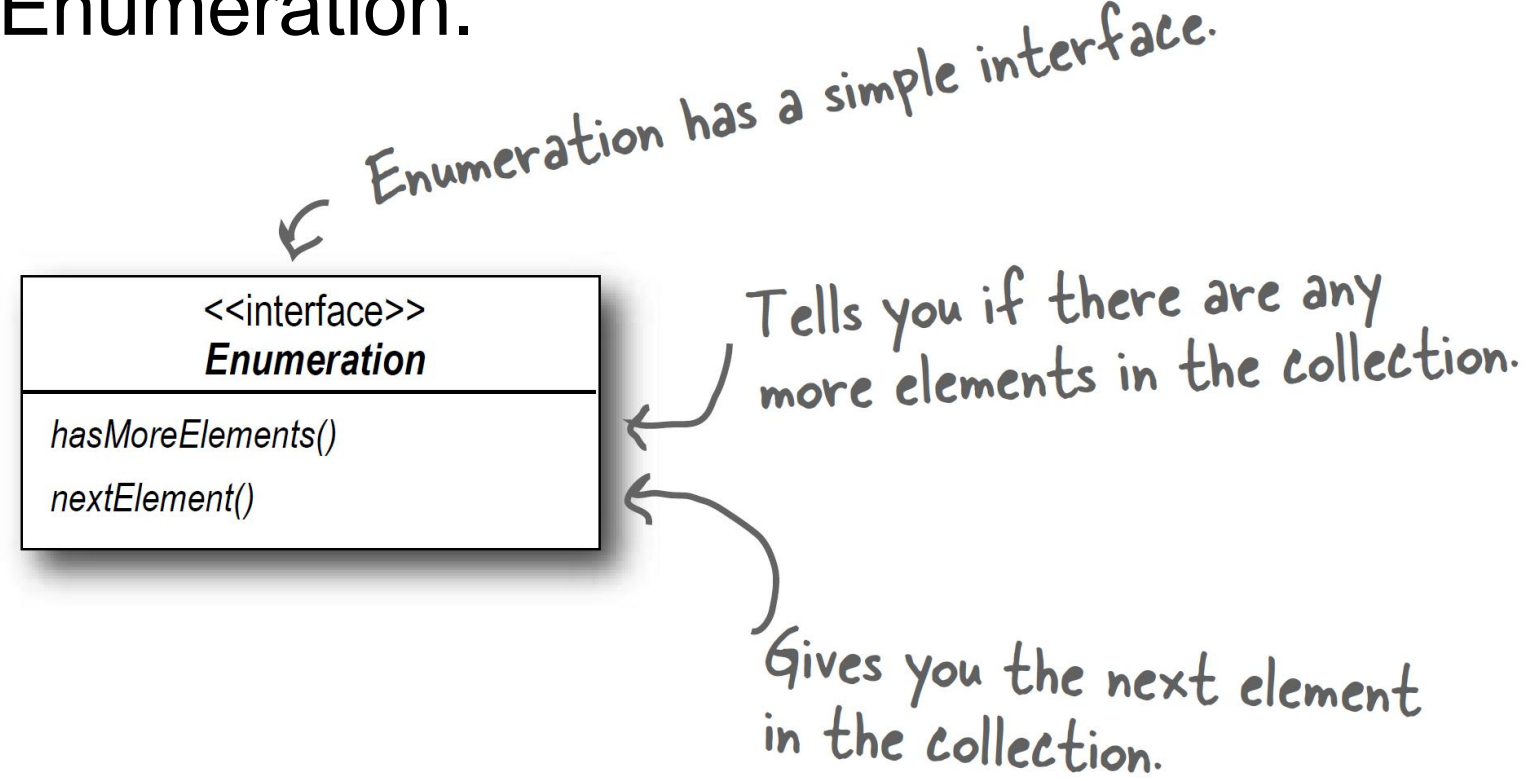
Consequences: Object Adapter

- A single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any).
- The Adapter can also add functionality to all Adaptees at once.
- It is harder to override Adaptee behavior.
  - It will require subclassing Adaptee, then
  - Making Adapter aggregated the subclass rather than the Adaptee itself.
- It is easy to add any new methods, what's more, the added method is suitable for all Adaptee.

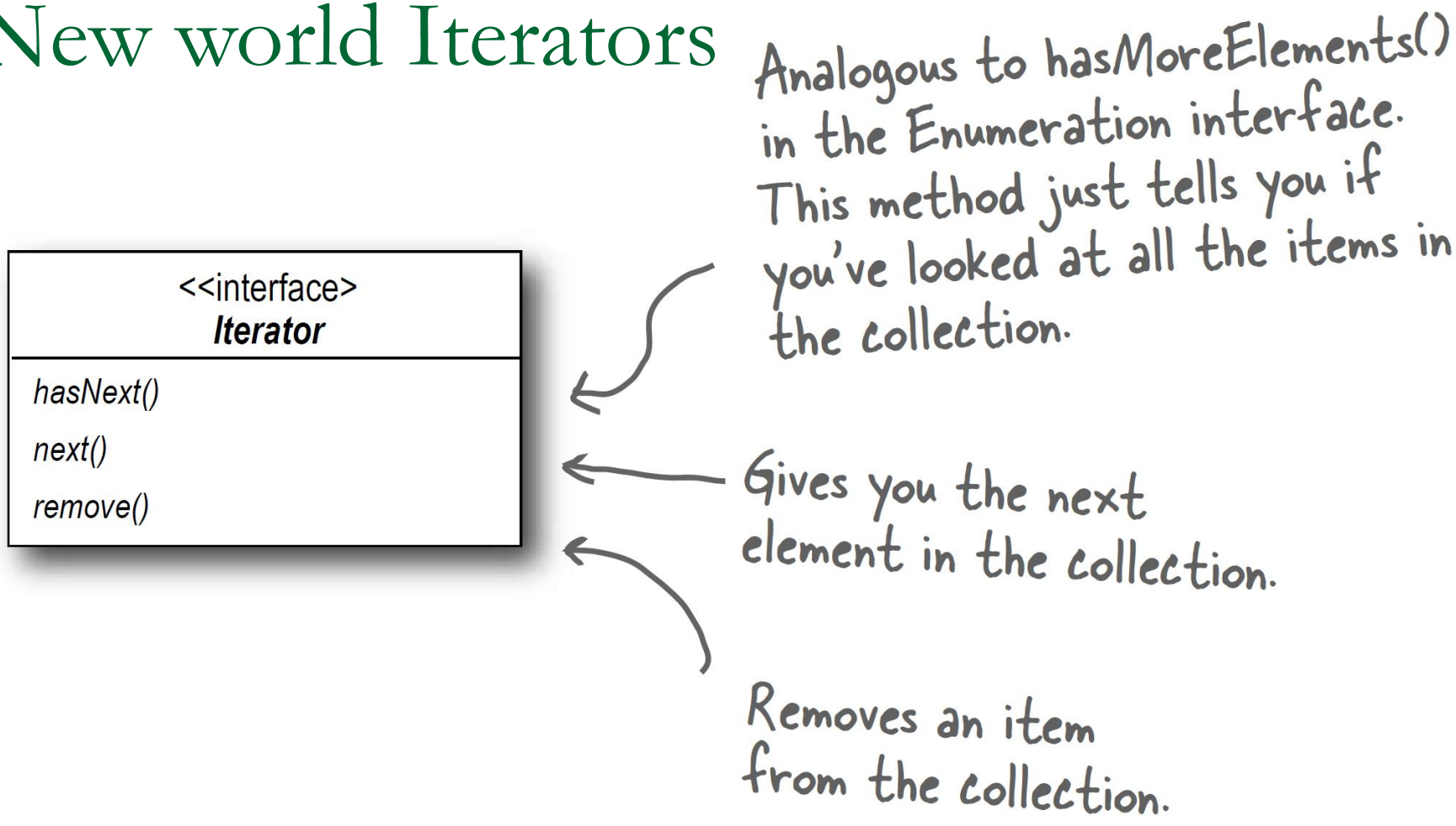
Real World Adapters: Enumeration and Iterator

Old world Enumerators

- The early collections types (Vector, Stack, Hashtable, and a few others) implement a method elements(), which returns an Enumeration.

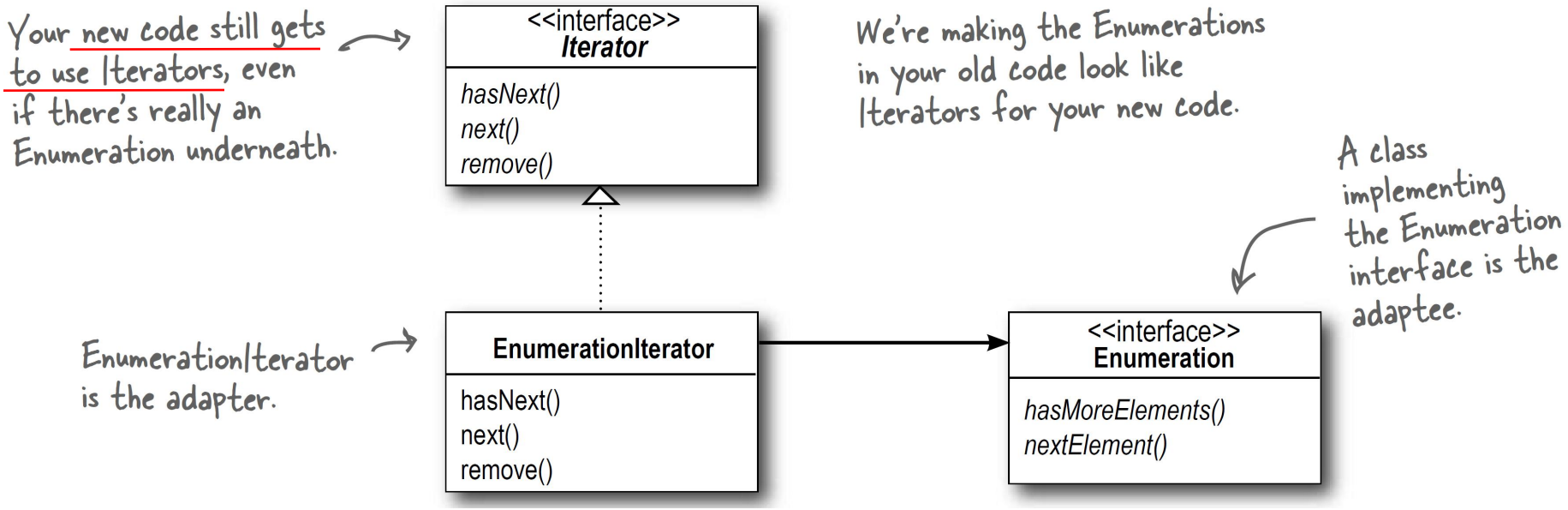


New world Iterators



- We are often faced with **legacy code** that exposes the Enumerator interface, yet we'd like for our new code to use only Iterators. It looks like we need to build an adapter.

Adapting an Enumeration to an Iterator: Designing the Adapter



Dealing with the remove() method

```
public class EnumerationIterator implements Iterator
{
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }

    public boolean hasNext() {
        return enum.hasMoreElements();
    }

    public Object next() {
        return enum.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Since we're adapting Enumeration to Iterator, our Adapter implements the Iterator interface... it has to look like an Iterator.

The Enumeration we're adapting. We're using composition so we stash it in an instance variable.

The Iterator's hasNext() method is delegated to the Enumeration's hasMoreElements() method...

... and the Iterator's next() method is delegated to the Enumerations's nextElement() method.

Unfortunately, we can't support Iterator's remove() method, so we have to punt (in other words, we give up!). Here we just throw an exception.