Head First Design Patterns

by Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates

ch07-01. Adapter Pattern

Adapter Pattern

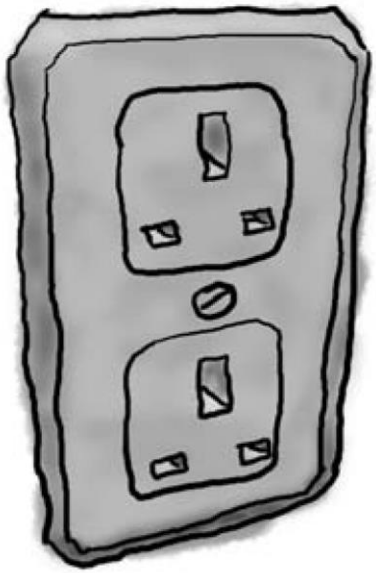
- **Also known as**
  - Wrapper
- **Purpose**
  - Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.
- **Use When**
  - A class to be used doesn't meet interface requirements.

The Adapter Pattern

- Motivation
  - A toolkit or class library may have an interface which is incompatible with an application's interface we want to integrate.
  - It is possible that we do not have access to the source code of the toolkit or library.
  - Even if the source code is available, we may want to minimize the change

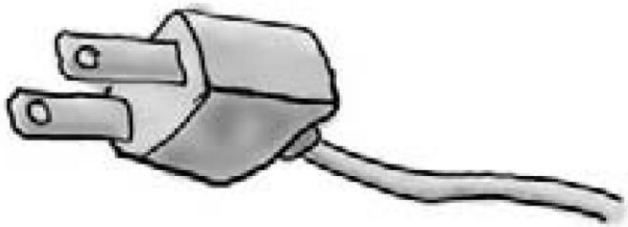
Adapters all around us

European Wall Outlet



AC Power Adapter

Standard AC Plug

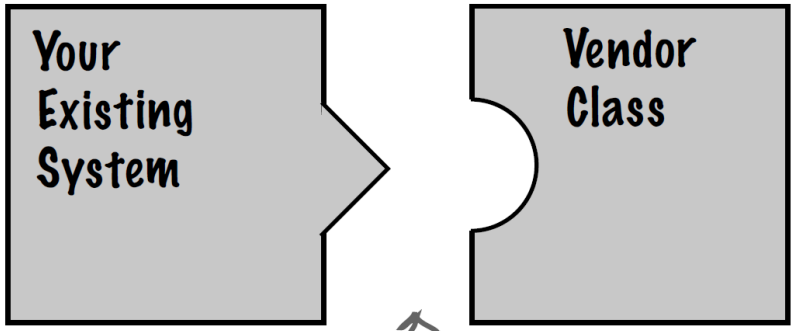


The US laptop expects another interface.

The European wall outlet exposes one interface for getting power.

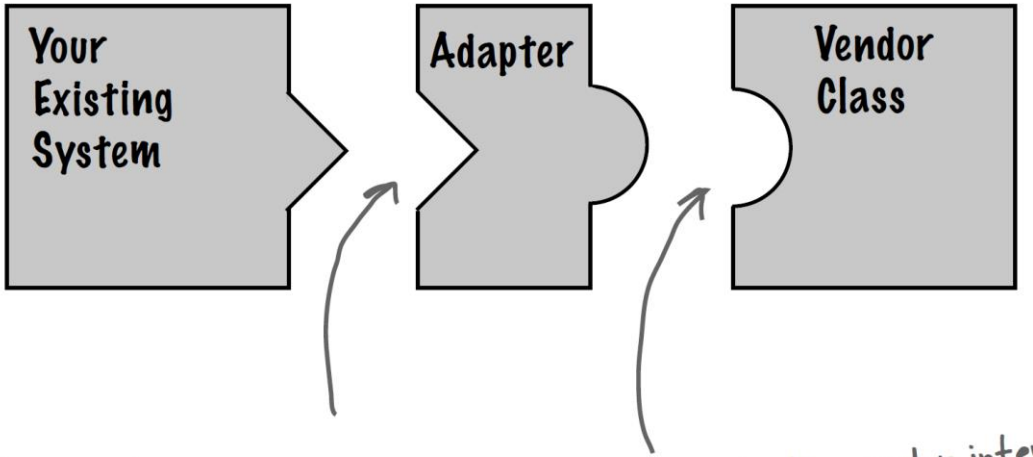
The adapter converts one interface into another.

Object oriented adapters



Their interface doesn't match the one you've written your code against. This isn't going to work!

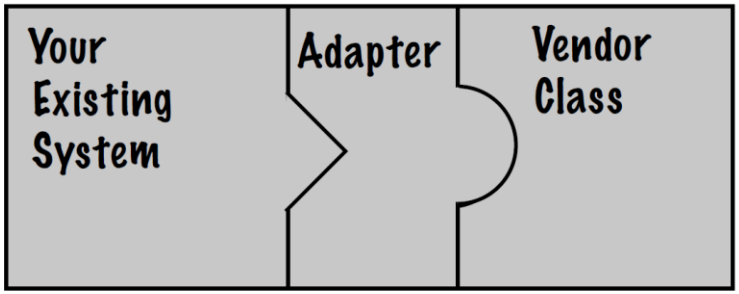
Object oriented adapters



The adapter implements the interface your classes expect.

And talks to the vendor interface to service your requests.

Object oriented adapters



No code changes.

New code.

No code changes.

Can you think of a solution that doesn't require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class.

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.

Now it's time to meet the newest fowl on the block:

```
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.

Now, let's say you're short on Duck objects and you'd like to use some Turkey objects in their place. Obviously we can't use the turkeys outright because they have a different interface.

Adapter

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

```
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();
        }
    }
}
```

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.

Test drive the adapter

```
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.

Test run

File Edit Window Help Don'tForgetToDuck

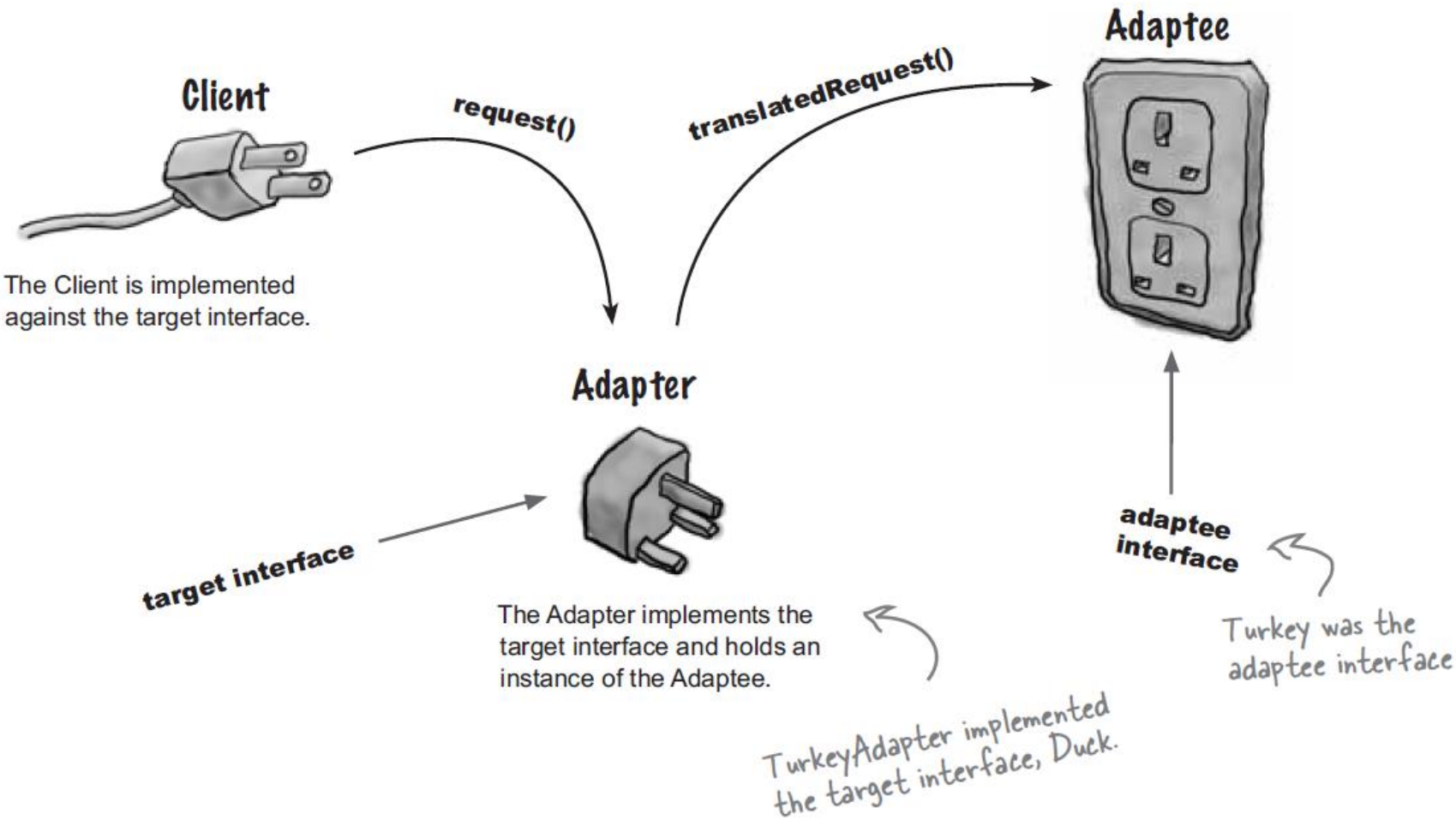
```
%java RemoteControlTest

The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

The Turkey gobbles and flies a short distance.

The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

The Adapter Pattern explained



Here's how the Client uses the Adapter

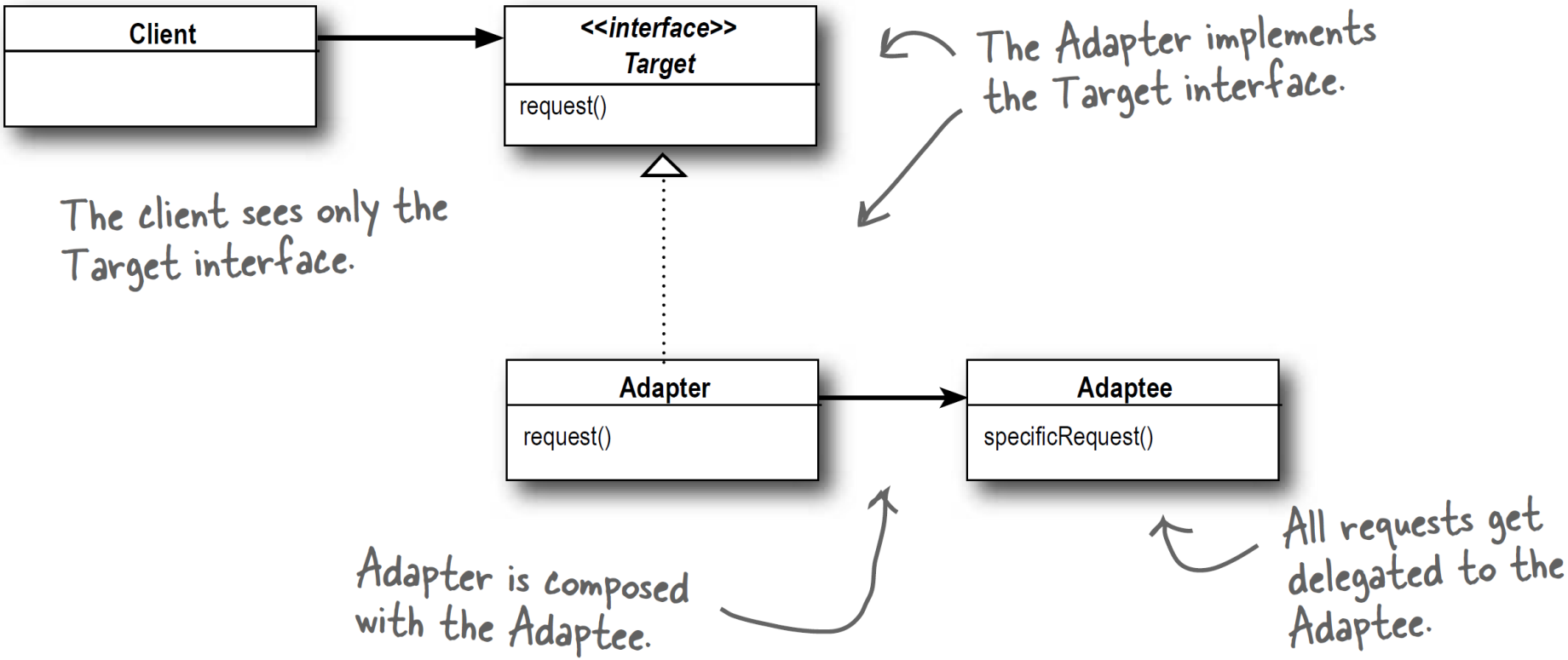
1. **The client makes a request to the adapter by calling a method on it using the target interface.**

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.**

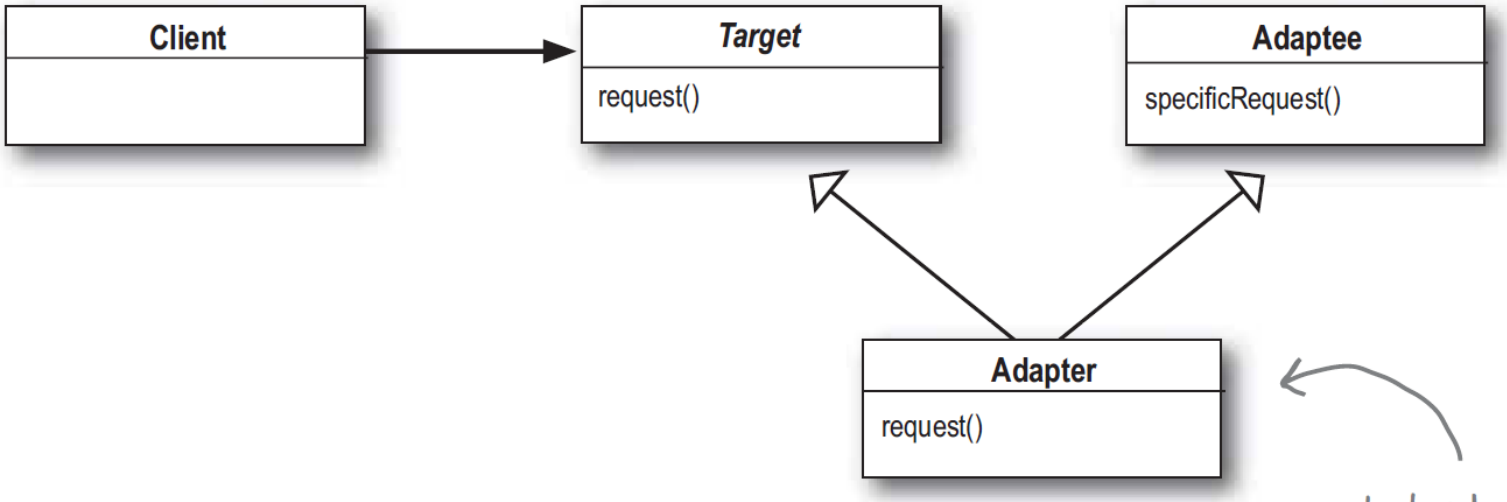
Note that the Client and Adaptee are decoupled – neither knows about the other.

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.


Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

Object and class adapters

Class Adapter

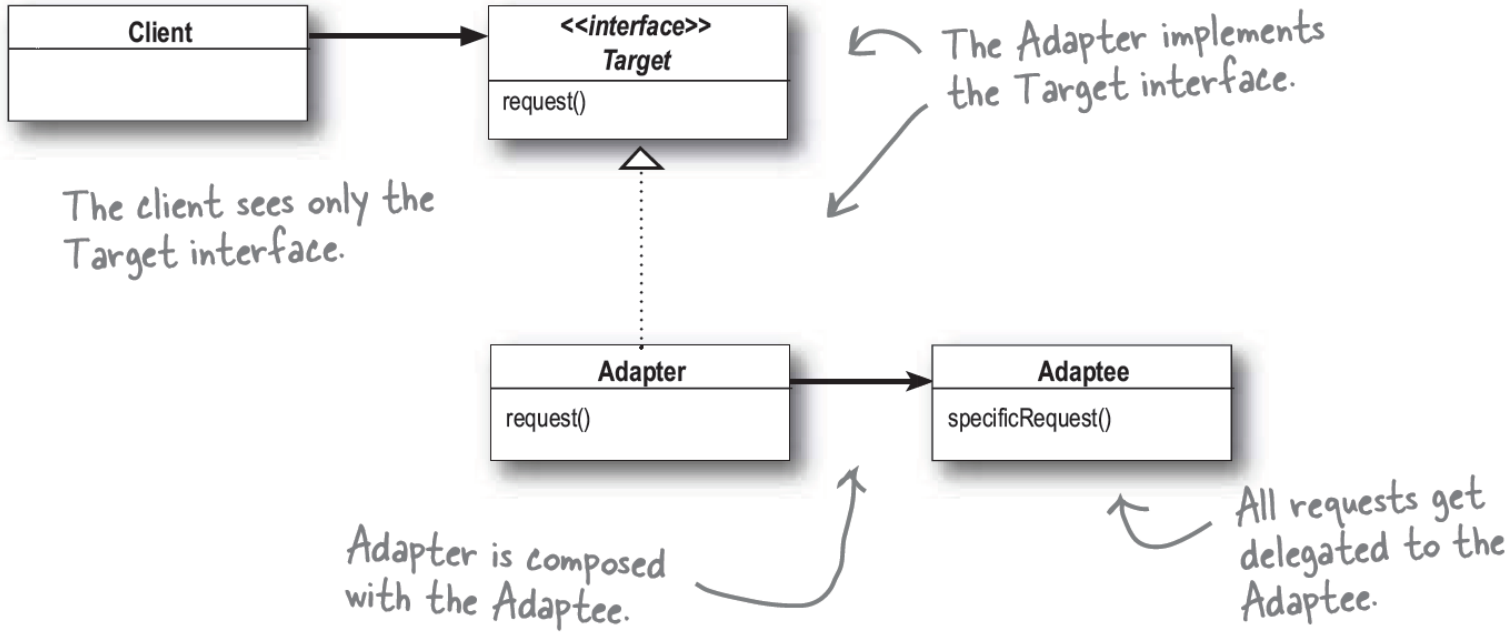


Instead of using composition to adapt the Adaptee, the Adapter now subclasses the Adaptee and the Target classes.

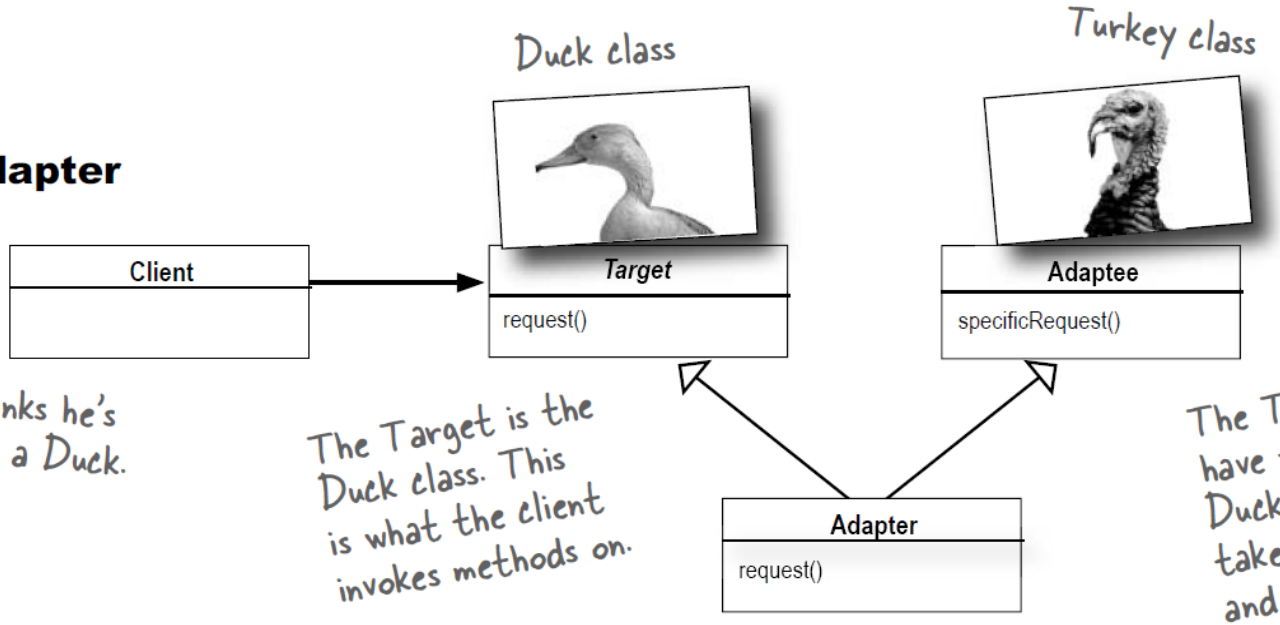
Object and class adapters

Object Adapter

- Use composition
- Program to interface



Class Adapter



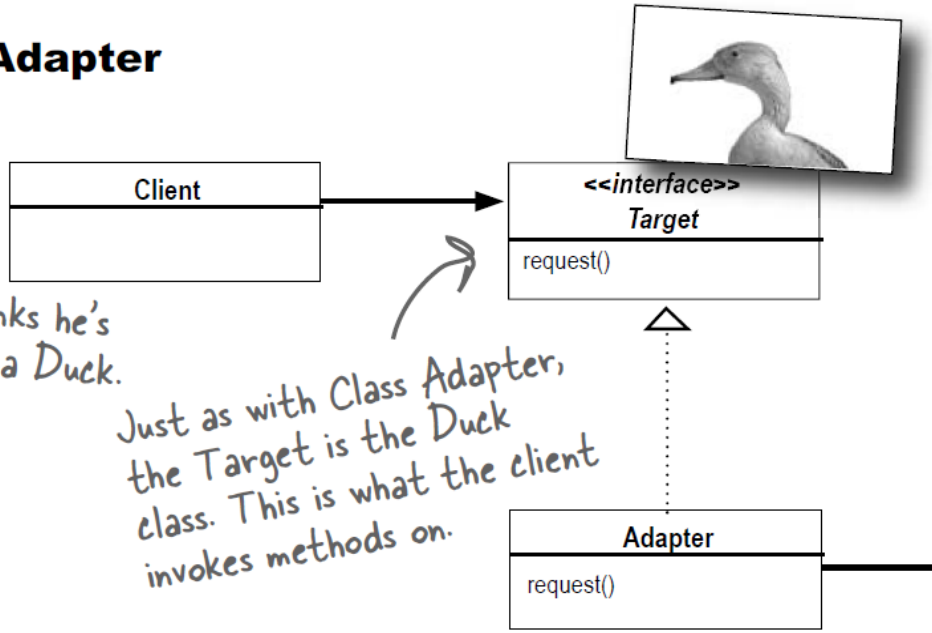
Client thinks he's talking to a Duck.

The Target is the Duck class. This is what the client invokes methods on.

The Adapter lets the Turkey respond to requests on a Duck, by extending BOTH classes (Duck and Turkey).

The Turkey class does not have the same methods as Duck, but the Adapter can take Duck method calls and turn around and invoke methods on the Turkey.

Object Adapter



Client thinks he's talking to a Duck.

Just as with Class Adapter, the Target is the Duck class. This is what the client invokes methods on.

The Turkey class doesn't have the same interface as the Duck. In other words, Turkeys don't have quack() methods, etc.

Turkey object.

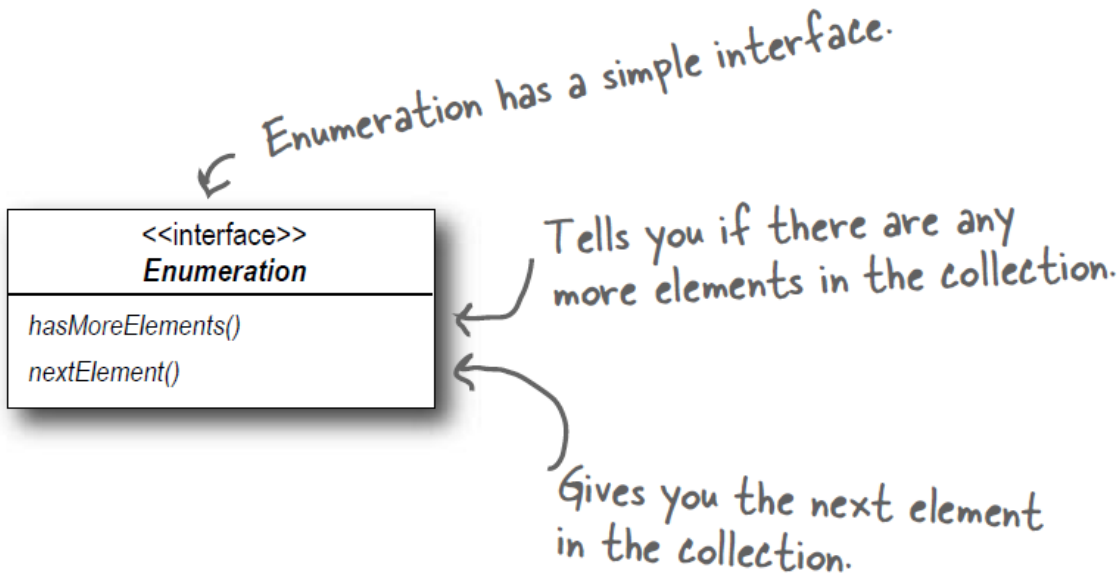
The Adapter implements the Duck interface, but when it gets a method call it turns around and delegates the calls to a Turkey.

Thanks to the Adapter, the Turkey (Adaptee) will get calls that the client makes on the Duck interface.

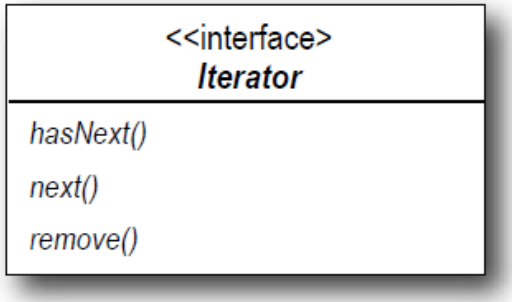
Real world adapters

Old world Enumerators

- Enumeration
  - Java's old collection types
    - Vector, Stack, HashTable, etc



New world Iterators

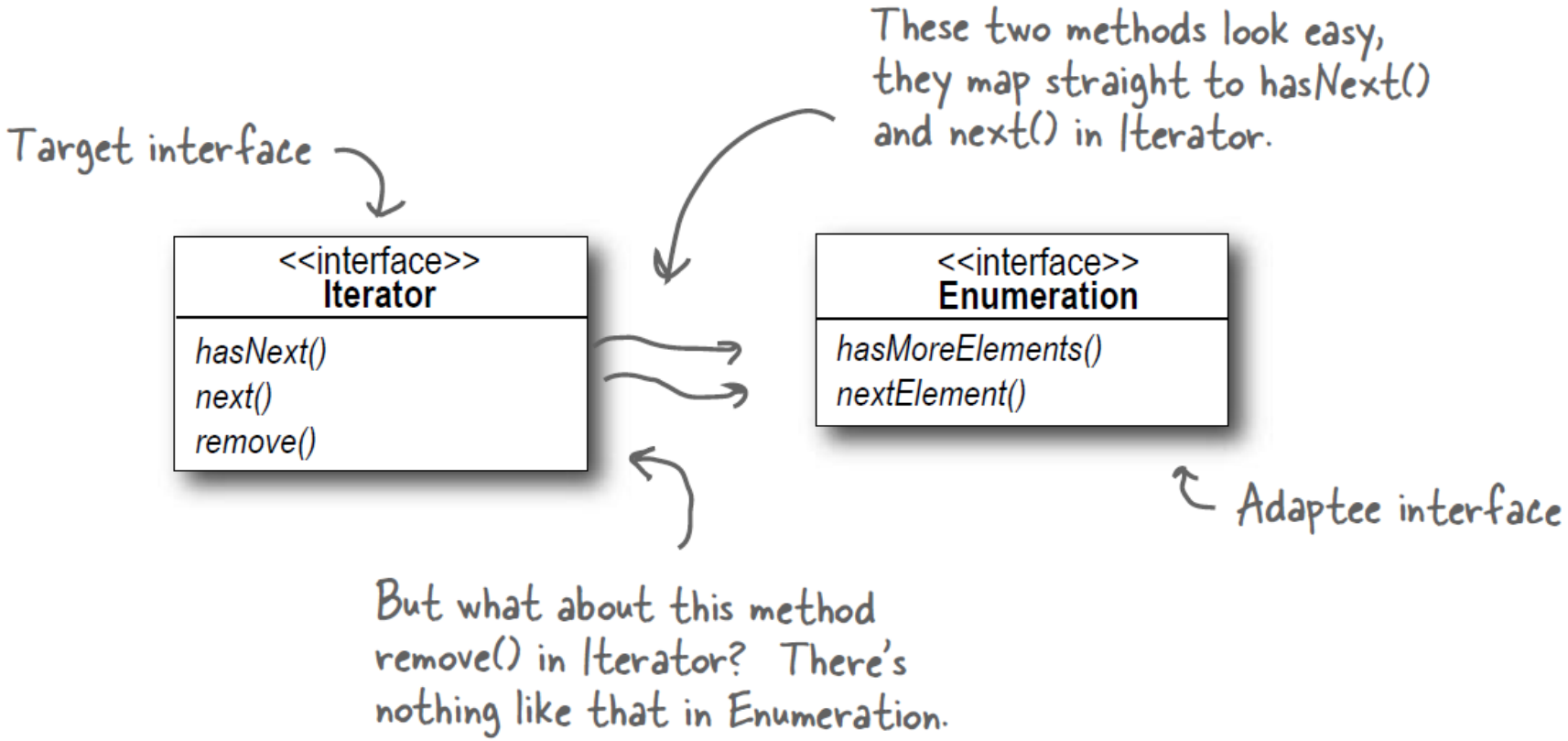


Analogous to hasMoreElements() in the Enumeration interface. This method just tells you if you've looked at all the items in the collection.

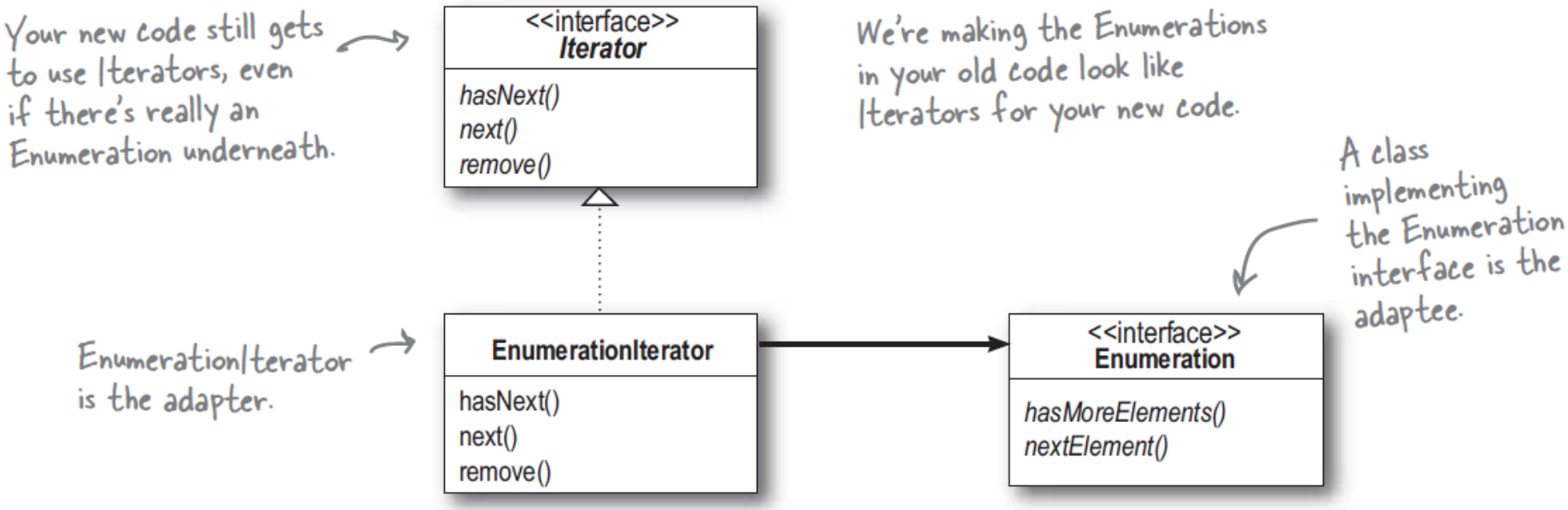
Gives you the next element in the collection.

Removes an item from the collection.

Adapting an Enumeration to an Iterator



Designing the Adapter



Writing the EnumerationIterator adapter

```
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.

IteratorEnumeration Adapter (Adapting Iterator to Enumeration)

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

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

    public Object nextElement() {
        return iterator.next();
    }
}
```