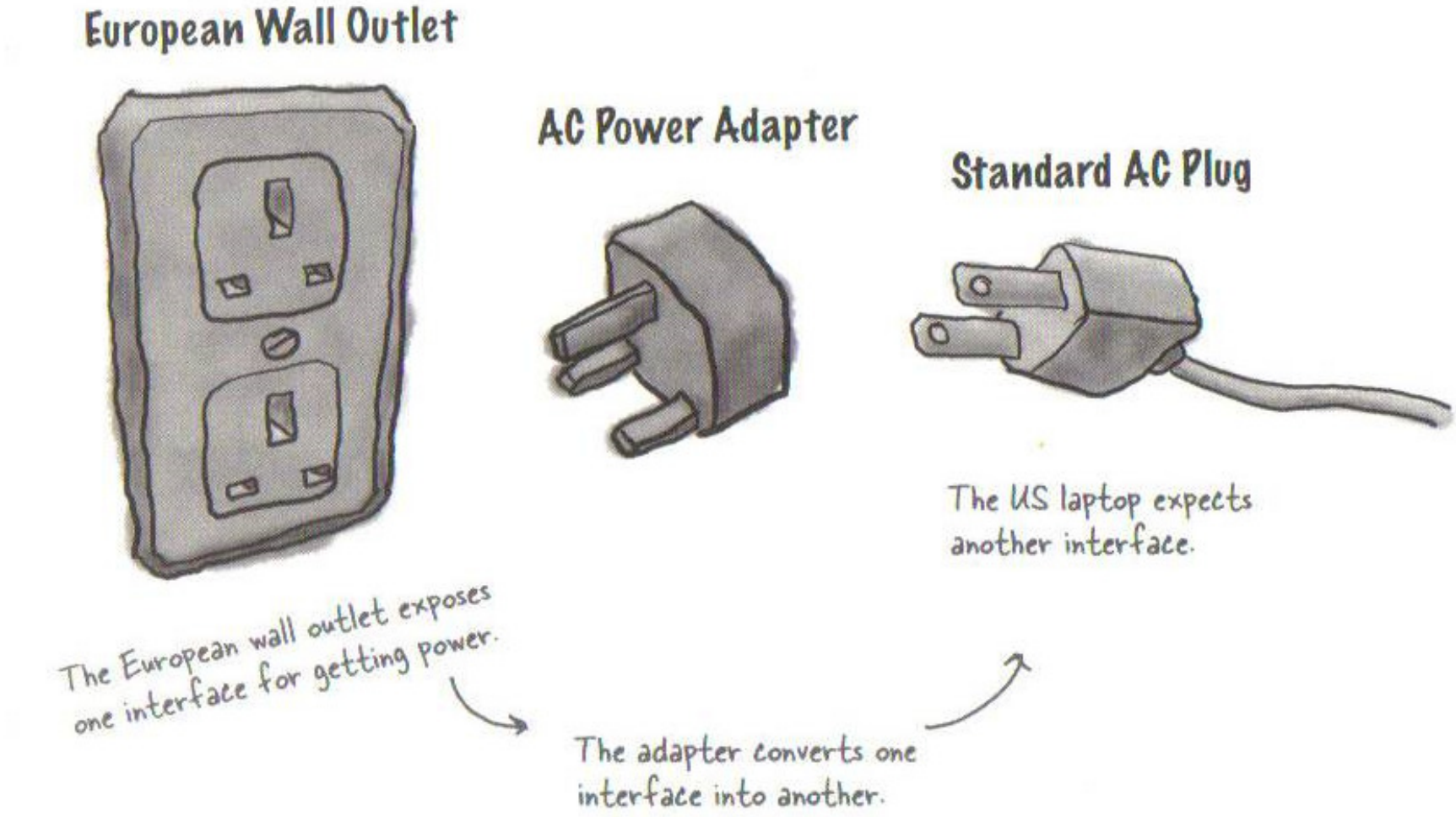
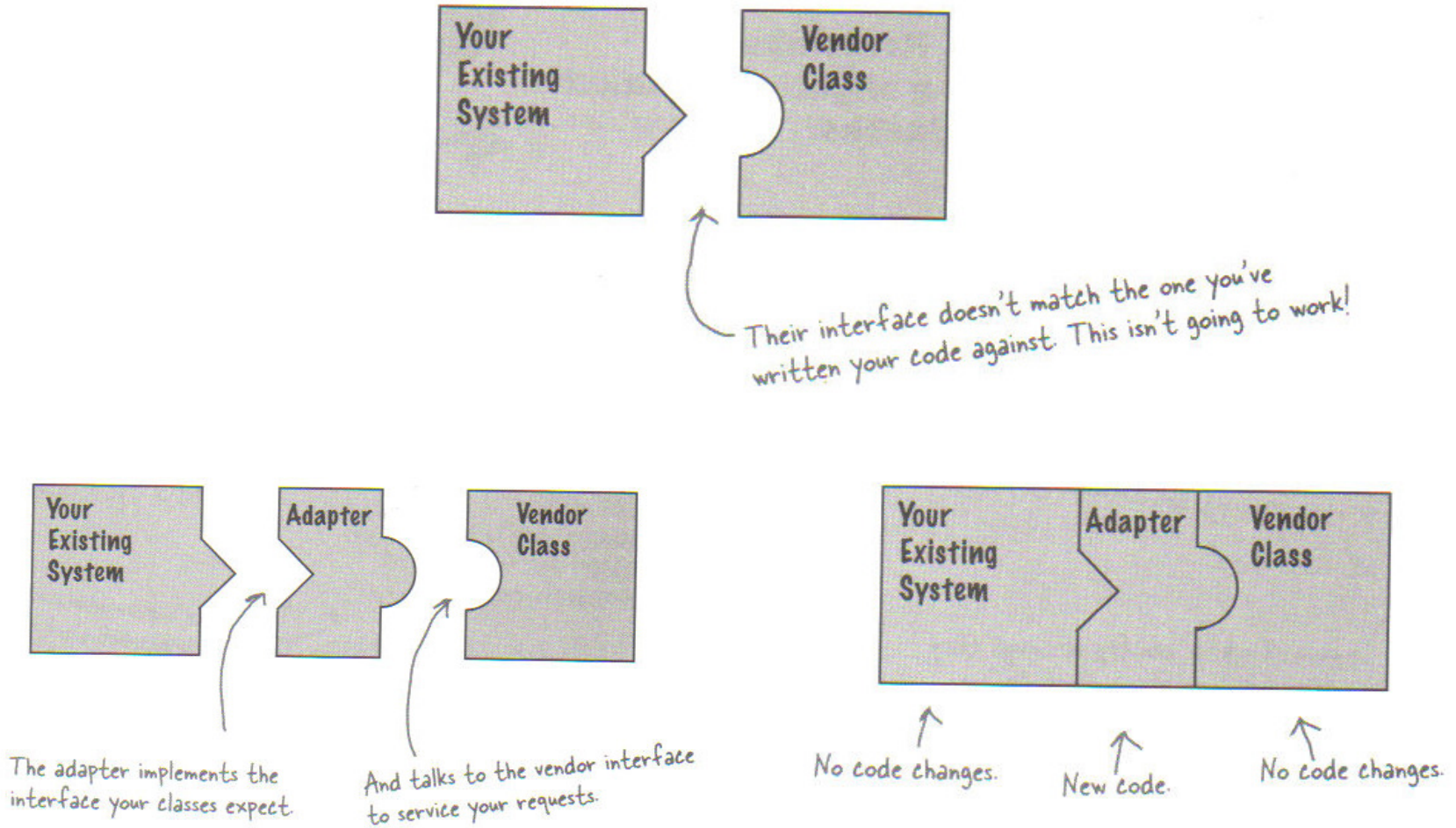
Tecniche di Progettazione: Design Patterns

GoF: Adapter

Adapters in real life (anglo-centric….)



Object-Oriented Adapters



Hugly Duckling example

```
public interface Duck {
    public void display();
    public void swim();
}
```

```
public interface Swan{
    public void show();
    public void swim();
}
```

```
public class Duckling implements Duck {
    public void display() {
       System.out.println(“I’m a pretty little
    duckling");
    }
    public void swim() {
            System.out.println("I'm learning…");
    }
}
```

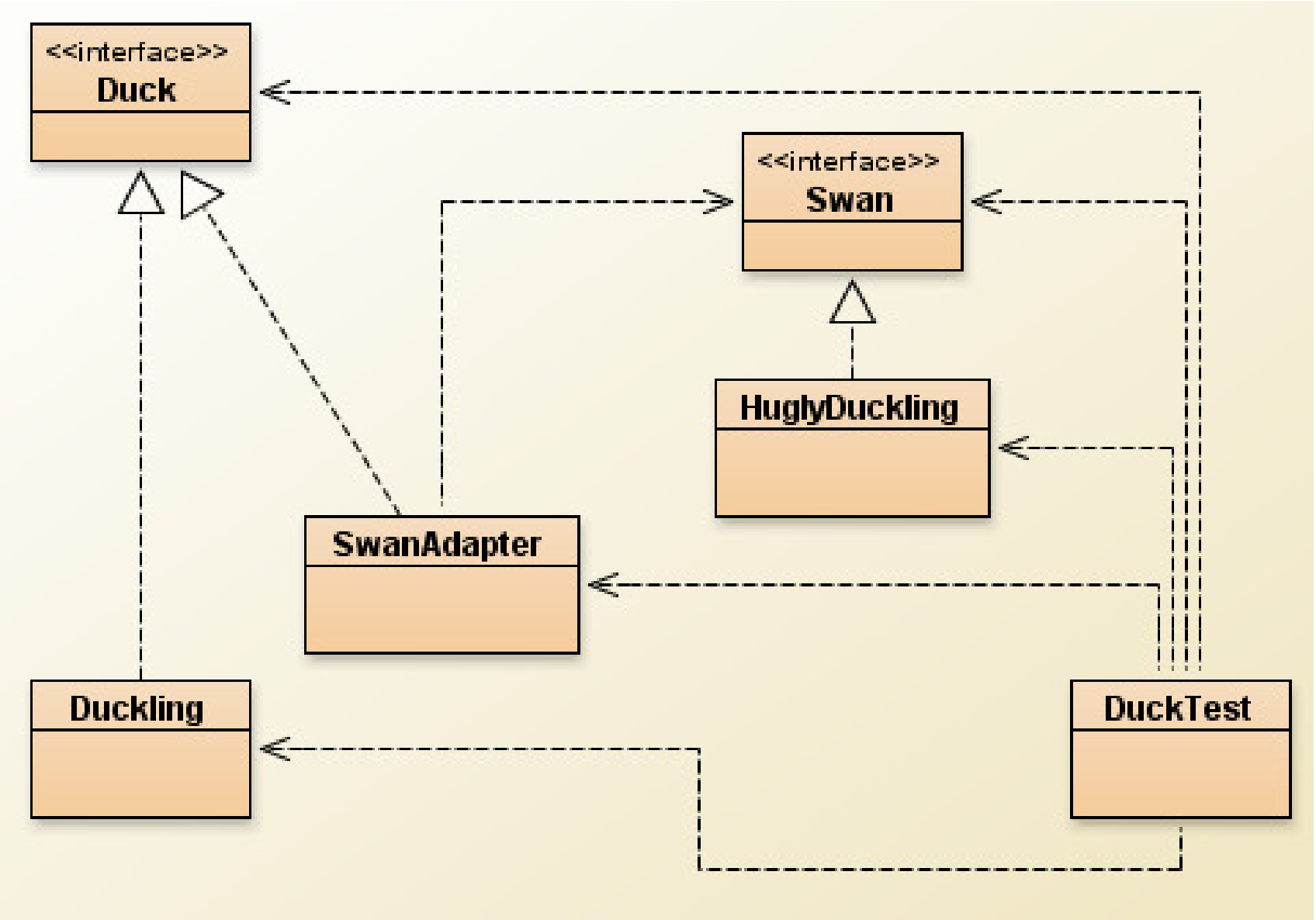
```
public class HuglyDuckling implements Swan{
    public void show() {
            System.out.println(“I’m large and hugly");
    }
    public void swim() {
            System.out.println("I'm swimming!’"");
    }
}
```

Adapter – that makes a swan (or turkey?) look like a duck

```
public class SwanAdapter implements Duck {
   Swan swan;
   public SwanAdapter(Swan swan) {
         this.swan = swan;
   }
         public void display() {
         swan.show ();
   }
   public void swim() {
         for(int i=0; i < 3; i++) {
                  swan.swim();
         }
   }
}
```

Duck test drive

```
public class DuckTest {
    public static void main(String[] args) {
            Duck duck = new Duckling();
            Swan hg= new HuglyDuckling();
            Duck  swanAdapter = new SwanAdapter(hg);
            System.out.println("The HuglyDuckling  says...");
            hg.show();
            hg.swim();
            System.out.println("\nThe Duck says...");
            testDuck(duck);
            System.out.println("\nThe SwanAdapter says...");
            testDuck(swanAdapter);
    }
    static void testDuck(Duck duck) {
            duck.display();
            duck.swim();
    }
}
```


<<interface>>
Duck
<<interface>>
Swan
HuglyDuckling
SwanAdapter
Duckling
DuckTest

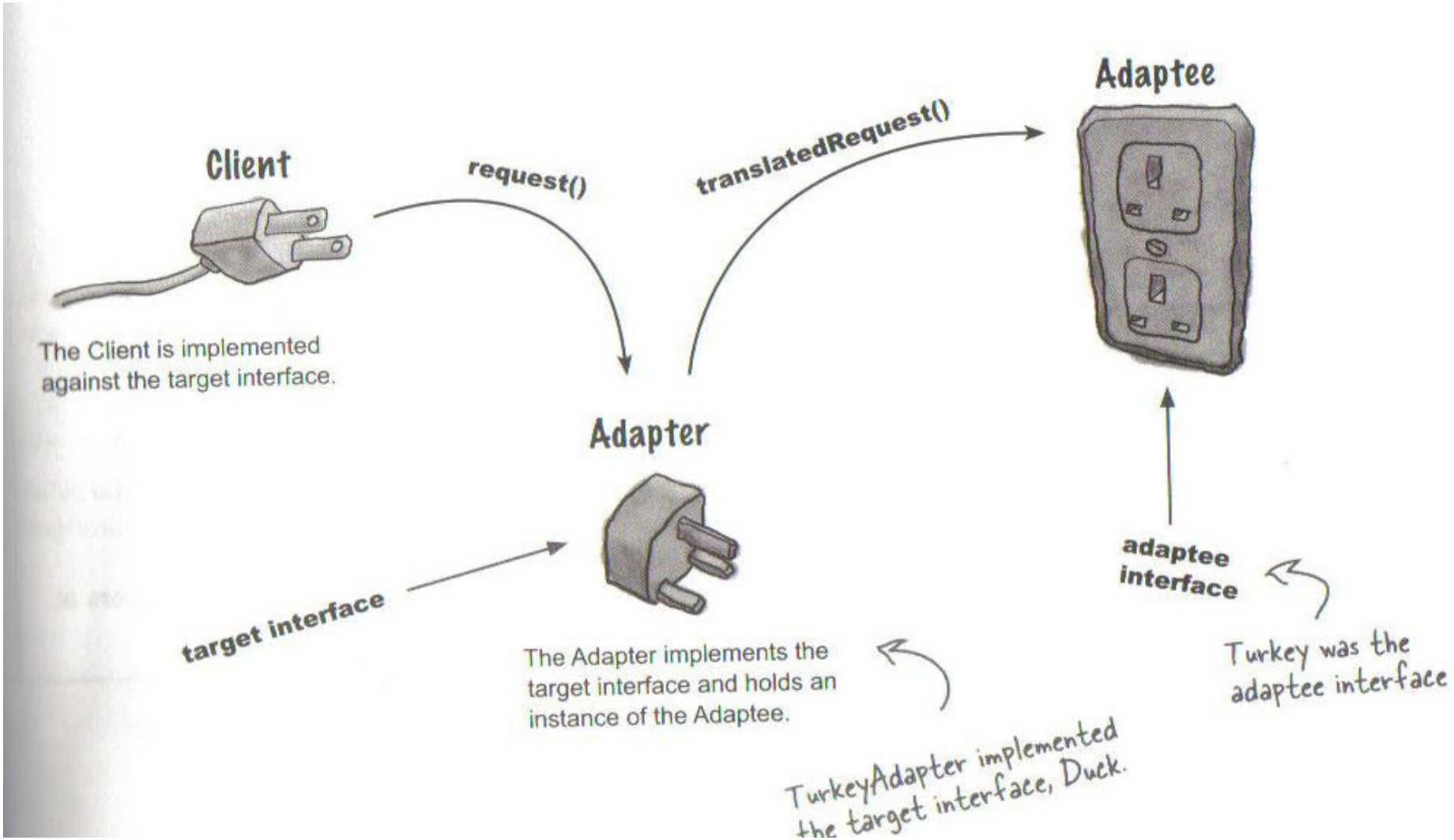
Test run – swan that behaves like a duck

The HuglyDuckling says...
I’m large and hugly
I'm swimming!”

The Duck says...
I'm a pretty little duckling
I'm learning…

The SwanAdapter says...
I’m large and hugly
I'm swimming!”
I'm swimming!”
I'm swimming!”

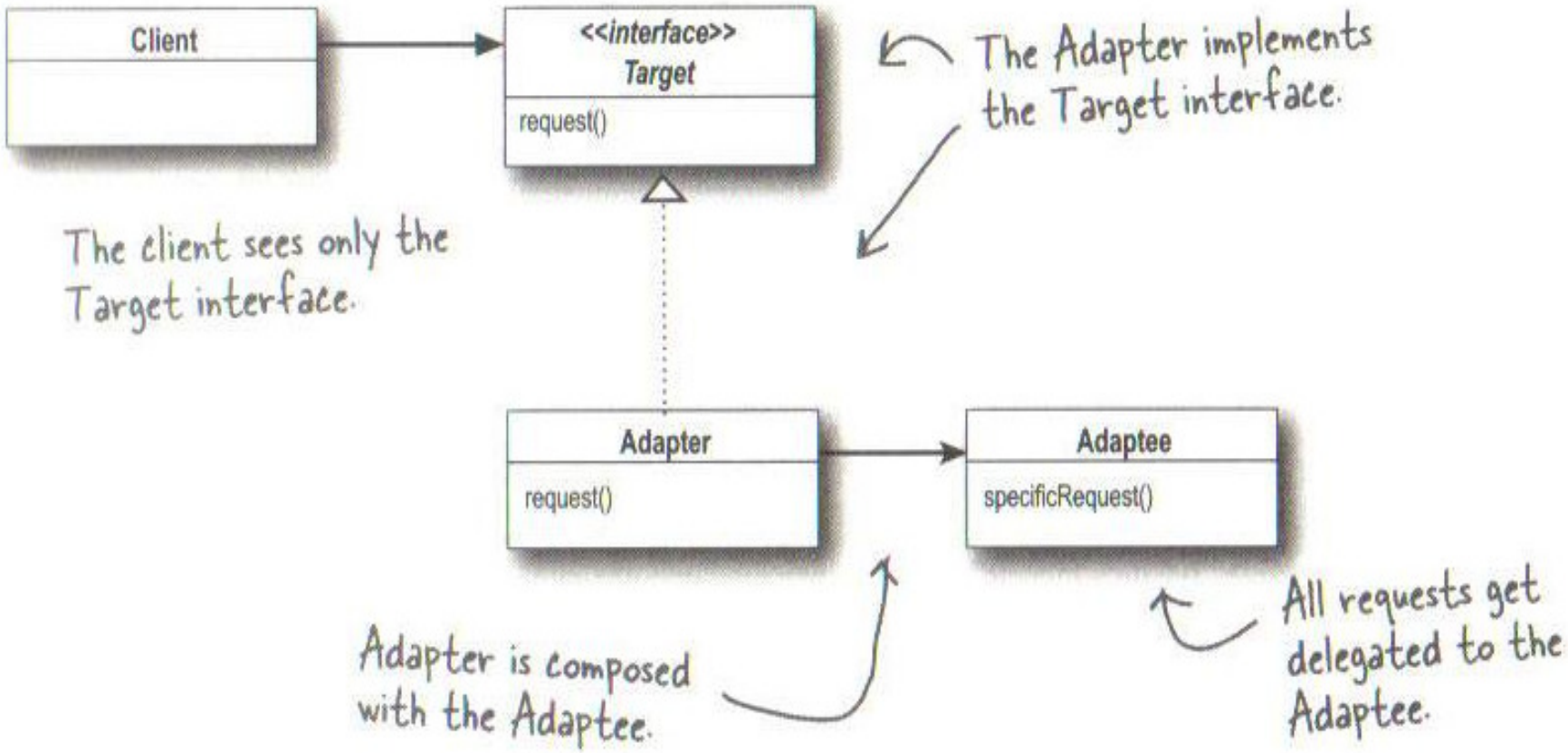
Adapter Pattern explained



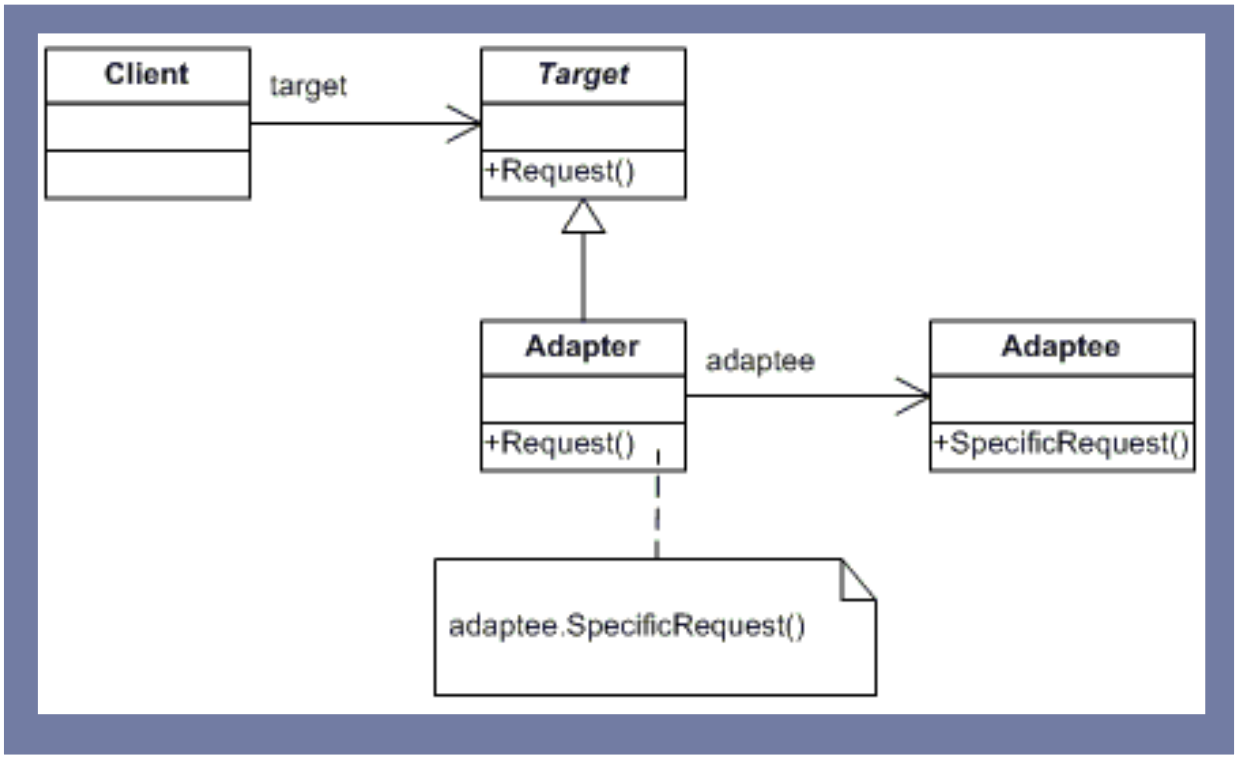
Adapter Pattern defined

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.

Adapter Pattern



Adapter pattern

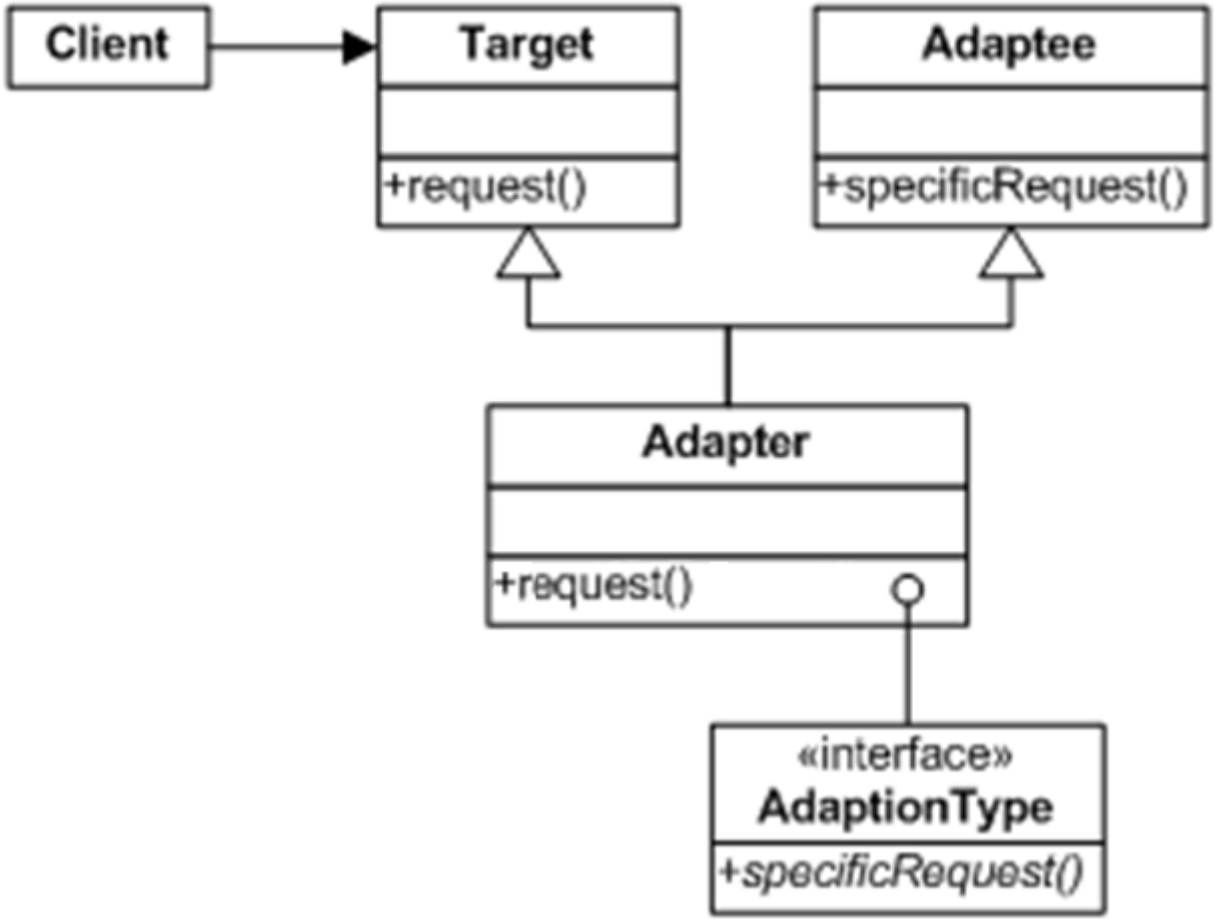


- Delegation is used to bind an Adapter and an Adaptee
- Interface inheritance is use to specify the interface of the Adapter class.
- Target and Adaptee (usually called legacy system) pre-exist the Adapter.
- Target may be realized as an interface in Java.

Participants

- **Target**: Defines the application-specific interface that clients use.
- **Client**: Collaborates with objects conforming to the target interface.
- **Adaptee**: Defines an existing interface that needs adapting.
- **Adapter**: Adapts the interface of the adaptee to the target interface.

Adapter with multiple inheritance



Two-way adapter

- What if we want to have an adapter that acts as a Target or an Adaptee?
- Such an adapter is called a two-way adapter.
  - One way to implement two-way adapters is to use multiple inheritance, but we can't do this in Java
  - But we can have our adapter class implement two different Java interfaces

Homework

- Define a differentiation as any feature or requirement that distinguishes one software system from another. When developing a family of similar products (e.g., mobile phones, university registration systems, video games), it is advantageous to identify the differentiations between products in or- der to develop efficient software component libraries that promote code reuse. Three basic types of differentiations have been identified:
  - Single differentiations are a set of mutually exclusive features, only one of which can be used in any given system. For example, all mobile phones have a display, but displays can vary (e.g., by the number of displayable characters).
  - Multiple differentiations are a set of optional features that are not mutually exclusive, where at least one is used in each system. For example, each mobile phone has at least one way to place a call, but there may be several (e.g., pressing the digits, pressing redial, voice dialing).
  - Optional differentiations are single features that may or may not be used. For example, mobile phones can have Internet connection capabilities, but they do not require them.
- Identify which of these three differentiation types can be effectively modeled using the Adapter pattern, and which cannot. Explain your responses.