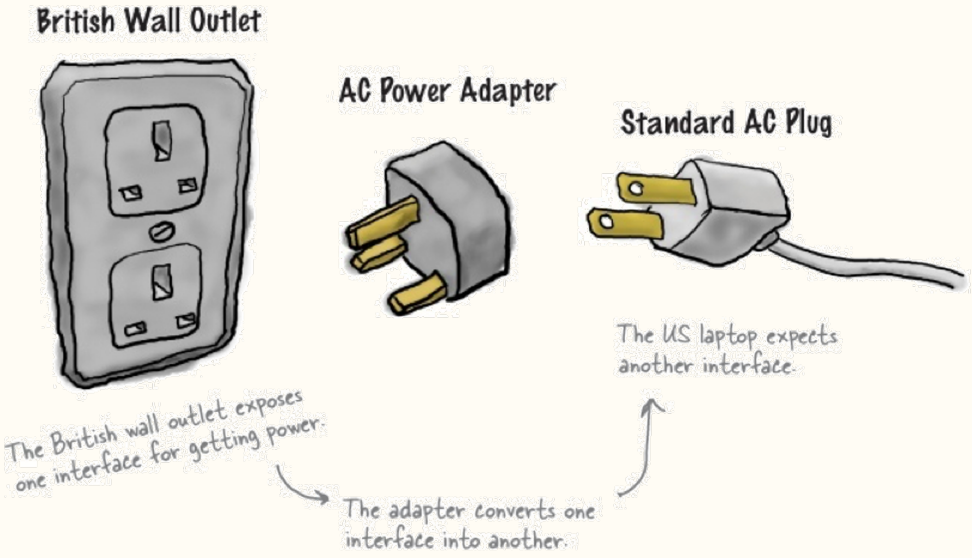
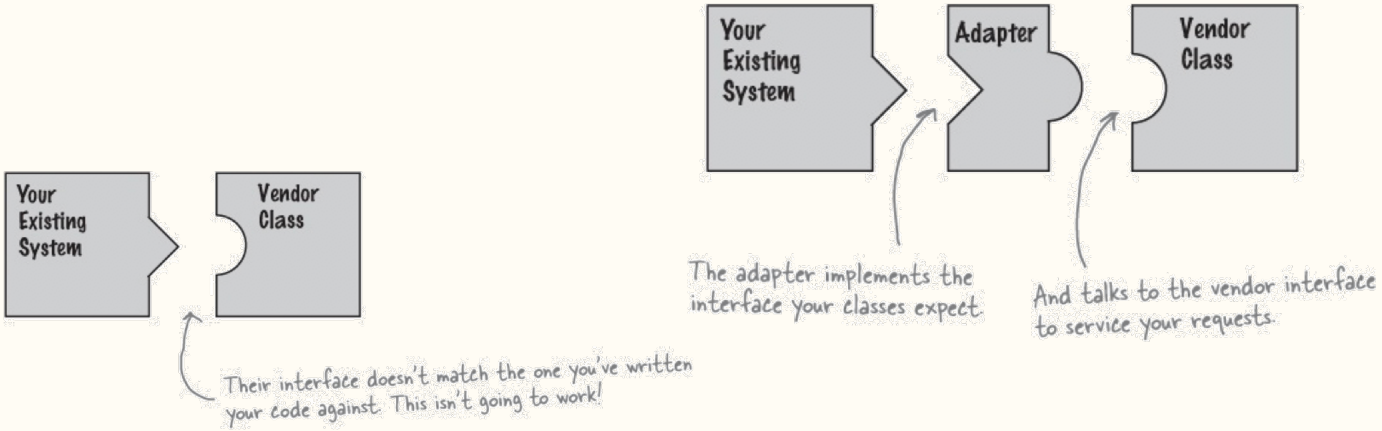
Adaptor pattern

Main Idea

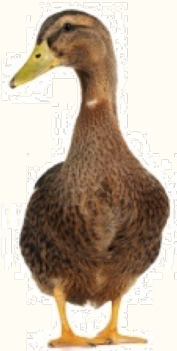
- Adaptation of one thing into another one
- Conversion of interface
- Indirect invocation of work



What's the adaptor in a SW?



Simplified Duck interfaces



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

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

In the "SimuDuck" simulation game!!!

Turkey interface and implementation

```
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.

<칠면조 수컷이>
Gobble: 골골 울다

```
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.

Adaptor

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

override 느낌

구현해준다 turkey에 있게.

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.

Test

```
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

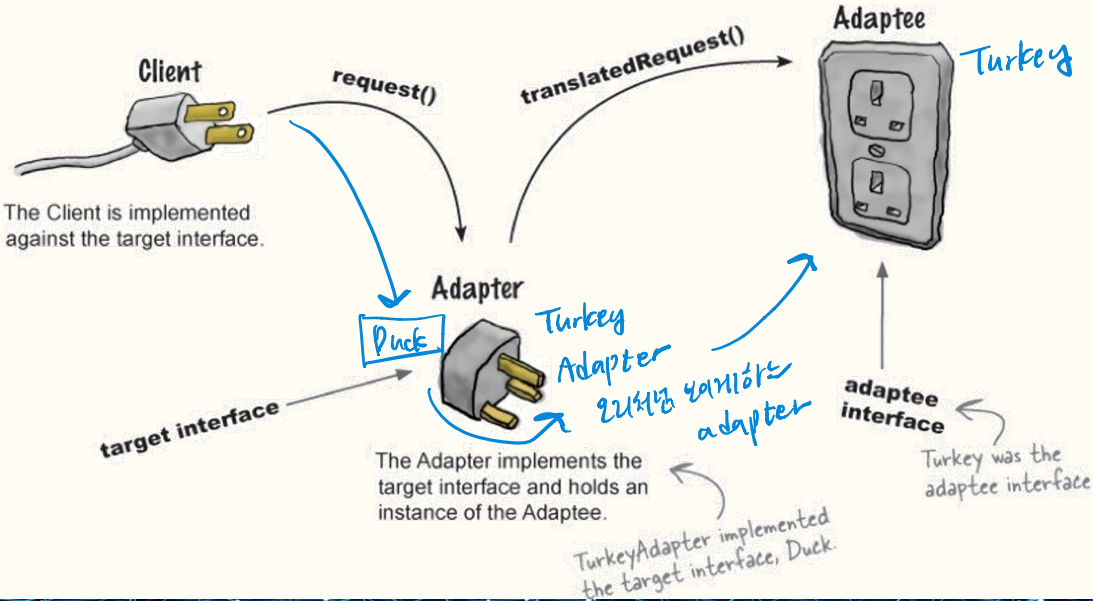
```
File Edit Window Help Don'tForgetToDuck
%java RemoteControlTest

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.

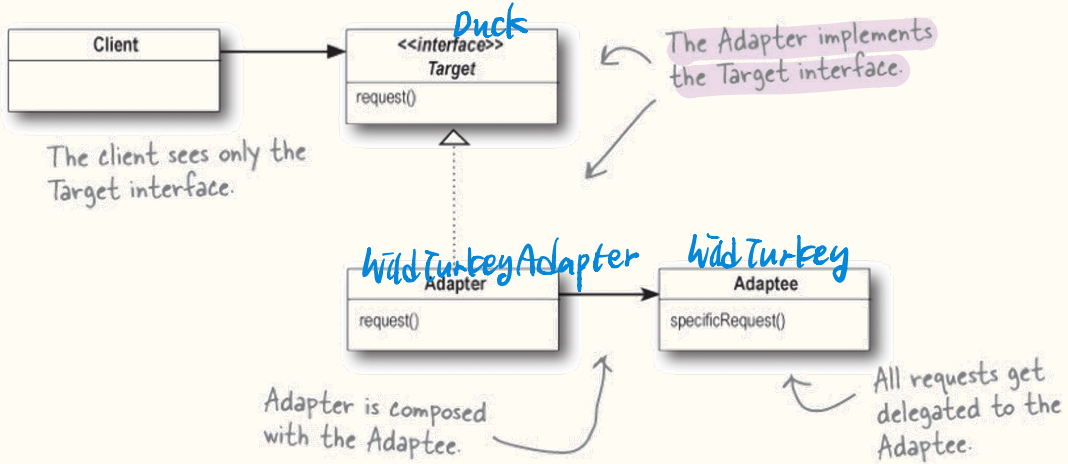
And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Mapping the idea into adaptor pattern

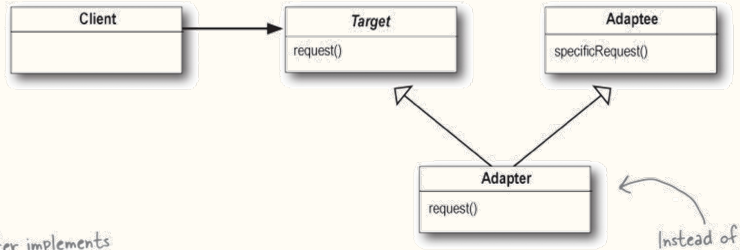


Definition of Adapter Pattern

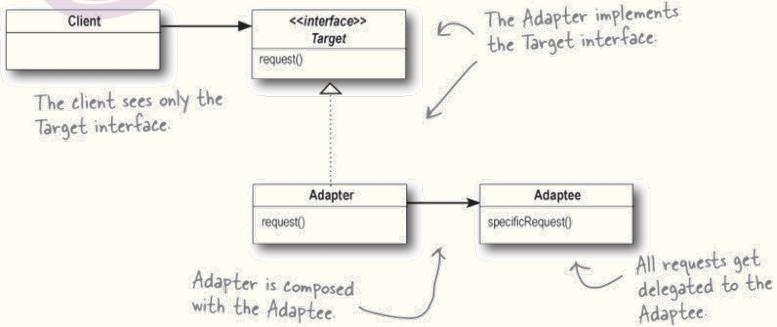
- Converts the interface of a class into another interface the clients expect.



Object and class adapter



Object adapter



Class adapter

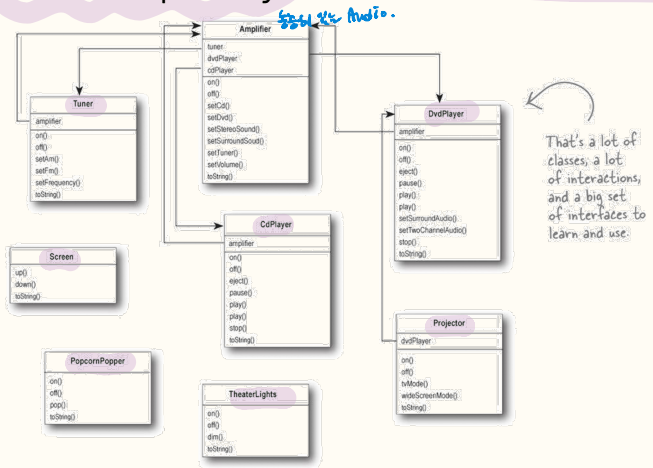
* 다중상속지원

⇓

class Turkey Adaptor implements Duck, extends Turkey 4ㄱ

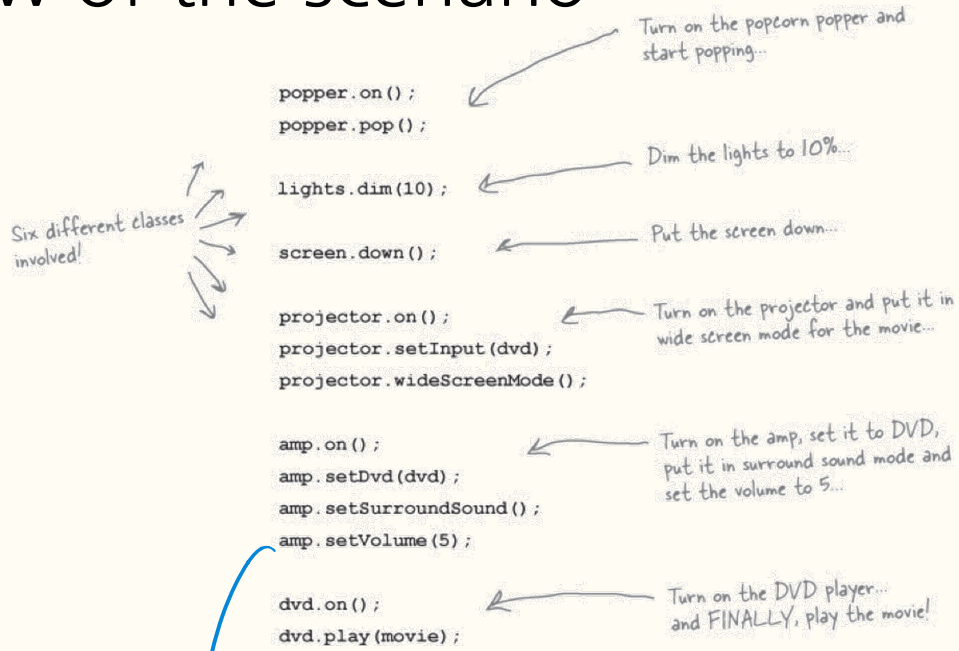
Façade Pattern 내부를 감추기 위함

- Alteration of an interface -> for simplification
  - Hides all the complexity of one or more class behind a clean, well-lit façade



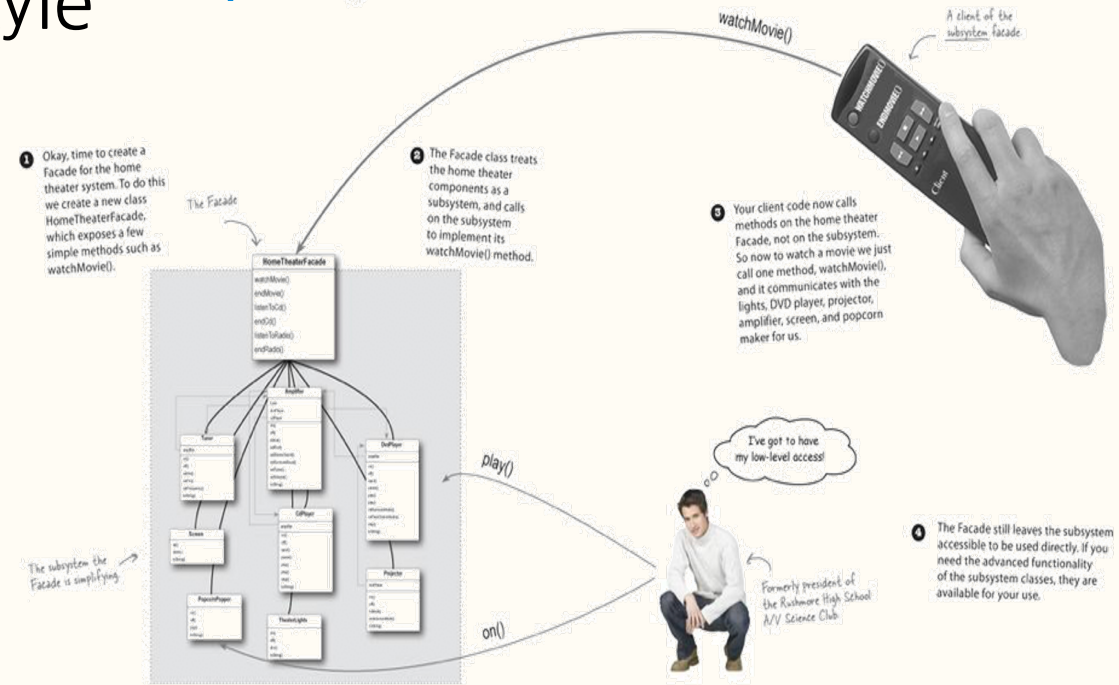
① Turn on the popcorn popper
② Start the popper popping
③ Dim the lights
④ Put the screen down
⑤ Turn the projector on
⑥ Set the projector input to DVD
⑦ Put the projector on wide-screen mode
⑧ Turn the sound amplifier on
⑨ Set the amplifier to DVD input
⑩ Set the amplifier to surround sound
⑪ Set the amplifier volume to medium (5)
⑫ Turn the DVD player on
⑬ Start the DVD player playing

Class View of the scenario



Façade Style

1. HomeTheaterFacade
   - Simplification
2. watchMovie() -> call subcomponents operations



HomeTheaterFacade class

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                             Tuner tuner,
                             DvdPlayer dvd,
                             CdPlayer cd,
                             Projector projector,
                             Screen screen,
                             TheaterLights lights,
                             PopcornPopper popper) {
        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    // other methods here

}
```

Here's the composition, these are all the components of the subsystem we are going to use.

parameter받아서 → 내것으로 setting.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

watchMovie()

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

watchMovie() follows the same sequence we had to do by hand before, but wraps it up in a handy method that does all the work. Notice that for each task we are delegating the responsibility to the corresponding component in the subsystem.

여러개의 method 호출
(↔ adapter 와 비슷)
↓
facade.

endMovie()


```
public void endMovie() {
    System.out.println("Shutting movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

And endMovie() takes care of shutting everything down for us. Again, each task is delegated to the appropriate component in the subsystem.

test

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }

}
```

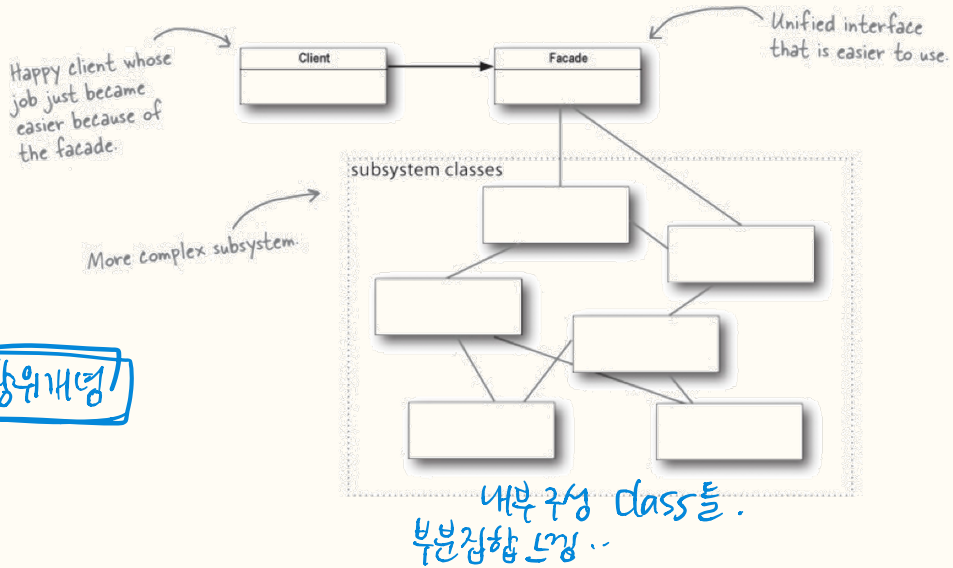
Here we're creating the components right in the test drive. Normally the client is given a facade; it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

Definition of Façade Pattern

- Provides a unified interfaced to a set of interfaces in a subsystem.
- Defines a higher-level interface that makes the subsystem easier to use



Principle of Least Knowledge

- Only invoke methods that belong to
  - The object itself
  - Objects passed in as a parameter to the method
  - Any object the method creates or instantiates

Do not make long indirect call path in an object

돌려받는 일이 아냐.